Formal Methods in System Design manuscript No.
(will be inserted by the editor)

Optimized Temporal Monitors for SystemC

Deian Tabakov · Kristin Y. Rozier · Moshe Y. Vardi

Received: date / Accepted: date

Abstract SystemC is a modeling language built as an extension of `C++`. Its growing popularity and the increasing complexity of designs have motivated research efforts aimed at the verification of SystemC models using assertion-based verification (ABV), where the designer asserts properties that capture the design intent in a formal language such as PSL or SVA. The model then can be verified against the properties using runtime or formal verification techniques. In this paper we focus on automated generation of runtime monitors from temporal properties. Our focus is on minimizing runtime overhead, rather than monitor size or monitor-generation time. We identify four issues in monitor generation: state minimization, alphabet representation, alphabet minimization, and monitor encoding. We conduct extensive experimentation and identify a combination of settings that offers the best performance in terms of runtime overhead.

Keywords SystemC, assertion checkers, monitors, testing

Work supported in part by NSF grants CCF-0613889, CCF-0728882, and Grant EIA-0216467, BSF grant 9800096, the Shared University Grid at Rice (SUG@R), NASA's Airspace Systems Program, a gift from Intel, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions.

A preliminary version of this work was reported by D. Tabakov and M.Y. Vardi in "Optimized temporal monitors for SystemC," Proc. 1st Int'l Conf. on Runtime Verification, Lecture Notes in Computer Science 6418, Springer, pp. 436–451, 2010.

Deian Tabakov
Schlumberger Information Solutions, 5599 San Felipe Str. #100, Houston, TX 77056, USA
E-mail: dtabakov@slb.com

Kristin Y. Rozier
NASA Ames Research Center, Moffett Field CA, 94035, USA
E-mail: Kristin.Y.Rozier@nasa.gov

Moshe Y. Vardi
Rice University, 6100 Main Str. MS-132, Houston, TX 77005, USA
E-mail: vardi@cs.rice.edu

1 Introduction

The increasing complexity of hardware designs and systems-on-chip (SoC), together with shortening timelines from prototype to mass production, have challenged the traditional RTL–based design procedures. A new paradigm was needed to allow modeling at higher levels of abstraction, gradual refinement of the model, and execution of the model during each design stage. SystemC[1] has emerged as one of the leading solutions of the "design gap."

SystemC is a system modeling language built as an extension of `C++`, providing libraries for modeling and simulation of systems on chip. It leverages the object-oriented encapsulation and inheritance mechanisms of `C++` to allow for modular designs and IP transfer/reuse [24]. Various libraries provide further functionality, for example, SystemC's Transaction-Level Modeling (TLM) library defines structures and protocols that streamline the development of high-level models. Thanks to its open-source license, actively involved community, and wide industrial adoption, SystemC has become a de facto standard modeling language within a decade after its first release.

Together, the growing popularity of SystemC and the increasing complexity of designs have motivated research efforts aimed at the verification of SystemC models using *assertion-based verification* (ABV), a widely used method for validation of hardware and software models [26]. With ABV, the designer asserts properties that capture design intent in a formal language, e.g., PSL[2] [17] or SVA[3] [45]. The model then is verified against the properties using runtime verification or formal verification techniques.

Most ABV efforts for SystemC focus on *runtime verification* (also called *dynamic verification*, *testing*, and *simulation*). This approach involves executing the model under verification (MUV) in some environment, while running monitors in parallel with the model. The monitors observe the inputs to the MUV and ensure that the behavior or the output is consistent with the asserted properties [24]. The complementary approach of *formal verification* attempts to produce a mathematical proof that the MUV satisfies the asserted properties. Our focus in this paper is on runtime verification.

A successful ABV solution requires two components: a formal declarative language for expressing properties, and a mechanism for checking that the MUV satisfies the properties. There have been several attempts to develop a formal declarative language for expressing temporal SystemC properties by adapting existing languages (see [42] for a detailed discussion). Tabakov et al. [42] argued that standard temporal property languages such as PSL and SVA are adequate to express temporal properties of SystemC models, after extending them with a set of Boolean assertions that capture the event-based semantics of SystemC. Enriching the Boolean layer, together with existing clock-sampling mechanisms in PSL and SVA, enables specification of proper-

[1] IEEE Standard 1666-2005

[2] *Property Specification Language*, IEEE Standard 1850-2007

[3] *SystemVerilog Assertions*, IEEE Standard 1800-2005

ties at different levels of abstraction. Tabakov and Vardi [41] then showed how a nominal change of the SystemC kernel enables monitoring temporal assertions expressed in the framework of [42] with overhead of about 0.05% – 1% per monitor. (Note that [41] used hand-generated monitors, while this work focuses on automatically generated monitors.)

The second component needed for assertion-based verification, a mechanism for checking that the MUV satisfies the asserted properties, requires a method for generating runtime monitors from formal properties. For simple properties it may be feasible to write the monitors manually (c.f., [20]); however, in most industrial workflows, writing and maintaining monitors manually would be an extremely high-cost, labor-intensive, and error-prone process [1]. This has inspired both academia and industry to search for methods to automate this process.

Formal, automata-theoretic foundations for monitor generation for temporal properties were laid out in [32], which showed how a deterministic finite word automaton (DFW) can be generated from a temporal property such that the automaton accepts the finite traces that violate the property. Many works have elaborated on that approach, c.f. [2, 3, 14, 18, 19, 23]); see the discussion below of related work. Many of these works, e.g. [2], handle only safety properties, which are properties whose failure is always witnessed by a finite trace. Here, as in [14], we follow the framework of [32] in its full generality and we consider all properties whose failure may be witnessed by a finite trace. For example, the failure of the property "eventually q" can never be witnessed by a finite trace, but the failure of the property "always p and eventually q" may be witnessed by a finite trace.

A priori it is not clear how monitor size is related to performance, and most works on this subject have focused on underlying algorithmics, or on heuristics to generate smaller monitors, or on fast monitor generation. This paper is an attempt to shift the focus toward optimizing the runtime overhead that monitor execution adds to simulation time. We believe that this reflects more accurately the priorities of the industrial applications of monitors [2].

A large model may be accompanied by thousands of monitors [5], most of which are compiled once and executed many times, so lower runtime overhead is a crucial optimization criterion, much more than monitor size or monitor-generation time. In this paper we identify several algorithmic choices that need to be made when generating temporal monitors for monitoring frameworks implemented in software. (Please note that here we ignore the issue of integrating the monitor into the monitored code; c.f. [41]). We conduct extensive experimentation to identify the choices that lead to superior performance.

We identify four issues in monitor generation: *state minimization*, should nondeterministic automata be determinized online or offline; *alphabet representation*, should alphabet letters be represented explicitly or symbolically; *alphabet minimization*, should mutually exclusive alphabet letters be eliminated; and *monitor encoding*, how should the transition function of the monitor be expressed. These options give us a *workflow space* of 33 different workflows for generating a monitor from a nondeterministic automaton.

We evaluate the performance of different monitor implementations using a SystemC model[4] representing an adder [41]. Its advantages are that it is scalable and creates events at many different levels of abstraction. For temporal properties we use linear temporal logic formulas. We use a mixture of pattern and random formulas, giving us a collection of over 1,300 temporal properties. We employ a tool called CHIMP (CHIMP Handles Instrumentation and Monitoring of Properties) to manage the transformation of LTL formulas into monitors using each of the 33 workflows. Our experiments identify a specific workflow that offers the best performance in terms of runtime overhead.

2 SystemC

Many contemporary systems consist of application-specific hardware and software, and tight production cycles make it impossible to wait for the hardware to be manufactured before starting to design the software. In a typical system-on-chip architecture [10], for example, a cell phone, there are hardware components that are controlled by software. In addition, many hardware design decisions, for example, numeric precision or the width of communication buses, are determined based on the needs of the software running on them. This has led to a design methodology where hardware and software are co-designed in the same abstract model. The partitioning between what will be implemented in hardware and what will be written as software is intentionally left blurry at the beginning, allowing the designers the ability to consider different configurations before committing a functional block to silicon or software.

SystemC is a system-level design framework that is capable of handling both hardware and software components. It allows a designer to combine complex electronic systems and control units in a single model, to simulate and observe the behavior, and to check if it meets the performance objectives. In the strict sense of the word, SystemC is not a new language. In fact, it is a library of `C++` classes and macros that model hardware components, like modules and channels; provide hardware-specific data types, like 4-valued logic types; and define both abstract and specific communication interfaces, like Boolean input. SystemC is built entirely on standard `C++`, which means that every SystemC model can be compiled with a `C++` compiler. The compiled model has to be linked with a SystemC simulator (for example, the OSCI-provided reference implementation) to produce an executable program.

Software typically executes sequentially, partly because most computer architectures have a single CPU core, and partly because a single thread of execution is easier to manage by the operating system. However, in a hardware system, many components execute simultaneously. For example, when using a cellphone to make a call, we activate simultaneously a radio subsystem that handles two-way communication with the cell tower, a signal processing unit that converts voice to signal and signal to voice, and a display controller that

[4] Note that the comparison is between different monitor implementations and is applicable to other `C` or `C++` modeling languages.

shows details about the conversation on the screen. Simulating such a system in software requires the ability to simulate a large number of tasks executing simultaneously, and is critical for the early stages of the design process.

SystemC addresses this issue by providing mechanisms for simulating (in software) parallel execution. This is achieved by a layered approach where high-level constructs share an efficient simulation engine [24]. The base layer of SystemC provides an event-driven simulation kernel that controls the model's processes in an abstract manner. The kernel leverages a concept borrowed from hardware design languages, called *delta cycles*, to give the executing processes the illusion of parallel execution.

In SystemC, modules are the most fundamental building blocks. Similar to C++ objects, modules allow related functionality and data to be incorporated into individual entities and to remain inaccessible by the other components of the system unless exposed explicitly. This allows modules to be developed independently and to be reused or sold in commercial libraries [8]. As an example, the skeleton of a SystemC module is presented in Listing 1:

```
1 SC_MODULE(Nand) {
2   // Definitions of processes, internal data, etc
3
4   SC_CTOR(Nand) {
5     // Body of constructor,
6     // Process registration,
7     // Definition of sensitivities, etc.
8   }
9 };
```

Listing 1 Skeleton code for defining a SystemC module.

In this code fragment, **SC_MODULE** is one of SystemC's macros, which declares a C++ class named "Nand." Like any other C++ class, a module can declare local variables and functions. **SC_CTOR** is another predefined macro that simplifies the definition of a constructor for the module. A constructor of a module serves the same purpose as a constructor of a C++ class (i.e., initializing local variables, executing functions, etc.), but has some additional functionality that is specific to SystemC. For example, the *processes* of the module have to be declared inside the constructor. This is done using predefined SystemC macros that specify which class functions should be treated by the SystemC kernel as runnable processes. After declaring each process, the user can optionally specify its *sensitivity list*. The sensitivity list may include a subset of the channels and signals defined in the module, as well as externally defined clock objects or events. Whenever there is a change of value of any of the channels or signals listed in the sensitivity list, the corresponding process is triggered for execution. Listing 2 illustrates these concepts.

```
1 SC_MODULE(Nand) {
2   // Input signal ports
3   sc_in <bool> A, B;
4   // Output signal port
5   sc_out<bool> F;
6
```

```
7    // Definitions of processes
8    void some_function() {
9      F.write(!(A.read() && B.read()));
10   }
11   SC_CTOR(Nand) {
12     // Indicate that this function
13     // is a ''method process''
14     SC_METHOD(some_function);
15     sensitive << A << B;
16   }
17 };
```

Listing 2 A SystemC module of a NAND gate

This code fragment declares one output and two input signals of type **bool**. The function some_function() implements the expected functionality of the NAND gate. The macro SC_METHOD declares it to be a SystemC process. When triggered, a *method process* executes from start to finish. In particular, a method process cannot suspend while waiting for some resource to become available. In contrast, a *thread process* may suspend its execution by calling **wait()**. The state of the thread process at the moment of suspension is preserved, and upon subsequent resumption (for example, when the waited-for resource becomes available) the execution continues from the point of suspension. Thread processes are declared using the macro **SC_THREAD**. Both thread and method processes can define a sensitivity list. Each sensitivity list declaration applies to the process immediately preceding the declaration. The sensitive declaration at the end of the module indicates that the method process some_function() should be triggered as soon as one of the input signals changes its value.

3 Related work

Most related papers that deal with monitoring focus on simplifying the monitor or reducing the number of states. Using smaller monitors is important for in-circuit monitoring, for example, for post-silicon verification [5], but for pre-silicon verification, using lower-overhead monitors is more important. There is a paucity of prior works focusing on minimizing runtime overhead.

For early work on constructing temporal monitors see [29]. Several papers focus on building monitors for *informative prefixes*, which are prefixes that violate input assertions in an "informative way." Kupferman and Vardi [32] define informative prefixes and show how to use an alternating automaton to construct a nondeterministic finite word automaton (NFW) of size $2^{O(\psi)}$ that accepts the informative prefixes of an LTL formula ψ. Kupferman and Lampert [31] use a related idea to construct an NFW automata of size $2^{O(\psi)}$ that accepts at least one prefix of every trace that violates a safety property ψ. Two constructions that build monitors for informative prefixes are by Geilen [19] and by Finkbeiner and Sipma [18]. Geilen's construction is based on the automata-theoretic construction of [22], while that of Finkbeiner and Sipma

is based on the alternating-automata framework of [32]. Neither provide experimental results.

Armoni et al. [2] describe an implementation based on [32] in the context of hardware verification. Their experimental results focus on both monitor size and runtime overhead. They showed that the overhead is significantly lower than that of commercial simulators. Stolz and Bodden [40] use monitors constructed from alternating automata to check specifications of Java programs, but do not give experimental results. For other works that focus on minimization see [4, 30, 33].

Giannakopoulou and Havelund [23] apply the construction of [22] to produce nondeterministic monitors for **X**–free LTL formulas, and simulate a deterministic monitor on the fly. They provide one experimental result from the early testing of their implementation. A weakness of their approach is that their LTL semantics distinguishes between finite and infinite traces, which implies that LTL properties may have different meanings in the context of dynamic and formal verification.

Morin-Allory and Borione [35] show how to construct hardware modules implementing monitors for properties expressed using the *simple subset* [25] of PSL. Pierre and Ferro [37] describe an implementation based on this construction, and present some experimental results that show runtime overhead, but do not present any attempts to minimize it. Boulé and Zilic [5] show a rewriting-based technique for constructing monitors for the simple subset of PSL. They provide substantial experimental results, but focus on monitor size and not on runtime overhead.

Chen et al. describe a general framework of Monitoring-Oriented Programming (MOP) [11]. In MOP, runtime monitoring is supported as a fundamental principle for building reliable software: monitors are automatically synthesized from specified properties and integrated into the original system to check its dynamic behaviors.

D'Amorim and Roşu [14] show how to construct monitors for *minimal bad prefixes* of temporal properties without any restrictions regarding whether the property is a safety property or not. They construct a nondeterministic finite automaton of size $2^{O(\psi)}$ that extracts the safety content from ψ, and simulate a deterministic monitor on the fly. They present two optimizations: one reduces the size of the automaton, while the other searches for a good ordering of the outgoing transitions so that the overall expected cost of running the monitor will be smallest. They measure experimentally the size of the monitors for a few properties, but do not measure their runtime performance. A similar construction, but without any of the optimizations, is also described by Bauer et al. [3].

4 Theoretical background

Let AP be a finite set of atomic propositions and let $\Sigma = 2^{AP}$ be a finite alphabet. Given a temporal specification ψ over AP, we denote the set of

models of the specification with $\mathcal{L}(\psi) = \{w \in \Sigma^\omega \mid w \models \psi\}$. Let $u \in \Sigma^*$ denote a finite word. We say that u is a *bad prefix* for $\mathcal{L}(\psi)$ iff $\forall \sigma \in \Sigma^\omega : u\sigma \notin \mathcal{L}(\psi)$ [32]. Intuitively, a bad prefix cannot be extended to an infinite word in $\mathcal{L}(\psi)$. A *minimal bad prefix* does not have a bad prefix as a strict prefix.

A *nondeterministic Büchi automaton* (NBW) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$, where Σ is a finite alphabet, Q is a finite set of states, $\delta : Q \times \Sigma \to 2^Q$ is a transition function, $Q^0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. If $q' \in \delta(q, \sigma)$ then we say that we have a transition from q to q' labeled by σ. We extend the transition function $\delta : Q \times \Sigma \to 2^Q$ to $\delta : 2^Q \times \Sigma^* \to 2^Q$ as follows: for all $Q' \subseteq Q$, $\delta(Q', a) = \cup_{q \in Q'} \delta(q, a)$, and for all $\sigma \in \Sigma^*$, $\delta(q, a\sigma) = \delta(\delta(q, a), \sigma)$. A *run* of $\mathcal{A}$ on a word $w = a_0 a_1 \ldots \in \Sigma^\omega$ is a sequence of states $q_0 q_1 \ldots$, such that $q_0 \in Q^0$ and $q_{i+1} \in \delta(q_i, a_i)$ for some $a_i \in \Sigma$. For a run r, let *Inf*(r) denote the states visited infinitely often. A run r of $\mathcal{A}$ is called *accepting* iff *Inf*$(r) \cap F \neq \emptyset$. The word w is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. For a given Linear Temporal Logic (LTL) or PSL/SVA formula ψ, we can construct an NBW that accepts precisely $\mathcal{L}(\psi)$ [44]. We use SPOT [16], an LTL-to-Büchi automaton tool, which is among the best available in terms of performance [39]. Using our framework for PSL or SVA would require an analogous translator.

A *nondeterministic automaton on finite words* (NFW) is a tuple $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$. An NFW can be determinized by applying the *subset construction*, yielding a *deterministic automaton on finite words* (DFW) $\mathcal{A}' = \langle \Sigma, 2^Q, \delta', \{Q^0\}, F' \rangle$, where $\delta'(Q, a) = \bigcup_{q \in Q} \delta(q, a)$ and $F' = \{Q : Q \cap F \neq \emptyset\}$. For a given NFW $\mathcal{A}$, there is a canonical minimal DFW that accepts $\mathcal{L}(\mathcal{A})$ [28]. In the remainder of this paper, given an LTL formula ψ, we use $\mathcal{A}_{NBW}(\psi)$ to mean an NBW that accepts $\mathcal{L}(\psi)$, and $\mathcal{A}_{NFW}(\psi)$ (respectively, $\mathcal{A}_{DFW}(\psi)$) to mean an NFW (respectively, DFW) that rejects the minimal bad prefixes of $\mathcal{L}(\psi)$.

Building a monitor for a property ψ requires building $\mathcal{A}_{DFW}(\psi)$. Our work is based on the construction by d'Amorim and Roşu [14], which produces $\mathcal{A}_{NFW}(\psi)$. Their construction assumes an efficient algorithm for constructing $\mathcal{A}_{NBW}(\psi)$ and is, therefore, is applicable to properties expressed in any a wide variety of specification languages (for example, if the property is expressed in LTL, $\mathcal{A}_{NBW}(\psi)$ can be constructed using [16]; for PSL specifications, the construction of $\mathcal{A}_{NBW}(\psi)$ can be done using [9]; etc.) Below we sketch the construction of [14] and then we show how we construct $\mathcal{A}_{DFW}(\psi)$.

Given an NBW $\mathcal{A} = \langle \Sigma, Q, \delta, Q^0, F \rangle$ and a state $q \in Q$, define $\mathcal{A}^q = \langle \Sigma, Q, \delta, q, F \rangle$. Intuitively, $\mathcal{A}^q$ is the NBW automaton defined over the structure of $\mathcal{A}$ but replacing the set Q^0 of initial states with $\{q\}$. Let *empty*$(\mathcal{A}) \subseteq Q$ consist of all states $q \in Q$ such that $\mathcal{L}(\mathcal{A}^q) = \emptyset$, i.e., all states that cannot start an accepting run. The states in *empty*$(\mathcal{A})$ are "unnecessary" in $\mathcal{A}$, because they cannot appear on an accepting run. We can compute *empty*$(\mathcal{A})$ efficiently using nested depth-first search [13]. Deleting the states in *empty*$(\mathcal{A})$ is done using the function call `spot::scc_filter()`, which is available in SPOT.

To generate a monitor for ψ, d'Amorim and Roşu build $\mathcal{A}_{NBW}(\psi)$ and remove $empty(\mathcal{A}_{NBW}(\psi))$. They then treat the resulting automaton as an NFW, with all states taken to be accepting states. That is, the resulting NFW is $\mathcal{A} = \langle \Sigma, Q', \delta', Q^0 \cap Q', Q' \rangle$, where $Q' = Q - empty(\mathcal{A})$, and δ' is δ restricted to $Q' \times \Sigma$. Let the automaton produced by this algorithm be $\mathcal{A}^{dR}_{NFW}(\psi)$.

Theorem 1 [14] *$\mathcal{A}^{dR}_{NFW}(\psi)$ rejects precisely the minimal bad prefixes of ψ.*

From now on we refer to $\mathcal{A}^{dR}_{NFW}(\psi)$ simply as $\mathcal{A}_{NFW}(\psi)$. $\mathcal{A}_{NFW}(\psi)$ is not useful as a monitor because of its nondeterminism. One way to construct a monitor from $\mathcal{A}_{NFW}(\psi)$ is to determinize it explicitly using the subset construction. In the worst case the resulting $\mathcal{A}_{DFW}(\psi)$ is of size exponential of the size of $\mathcal{A}_{NFW}(\psi)$, which is why explicit determinization has rarely been used. We note, however, that we can minimize $\mathcal{A}_{DFW}(\psi)$, getting a minimal DFW. It is not clear, a priori, what impact this determinization and minimization will have on runtime overhead.

An alternative way of constructing a monitor from $\mathcal{A}_{NFW}(\psi)$ that avoids the potential for exponential blow up of the number of states is to use $\mathcal{A}_{NFW}(\psi)$ to simulate a deterministic monitor on the fly. d'Amorim and Roşu describe such a construction in terms of nondeterministic multi-transitions and binary transition trees [14]. Instead of introducing these formalisms, here we use instead the approach in [2, 43], which presents the same concept in automata-theoretic terms. The idea in both papers is to perform the subset construction on the fly, as we read the inputs from the trace. Given $\mathcal{A}_{NFW}(\psi) = \langle \Sigma, Q, \delta, Q^0, Q \rangle$ and a finite trace $a_0, \ldots, a_{n-1}$, we construct a run $P_0, \ldots, P_n$ of $\mathcal{A}_{DFW}(\psi)$ as follows: $P_0 = \{Q^0\}$ and $P_{i+1} = \bigcup_{s \in P_i} \delta(s, a_i)$. The run is accepting iff $P_i = \emptyset$ for some $i \geq 0$ (i.e., no transition is enabled), which means that we have read a bad prefix. Notice that each P_i is of size linear in the size of $\mathcal{A}_{NFW}(\psi)$, thus we have avoided the exponential blowup of the determinization construction, with the price of having to compute transitions on the fly [2, 43].

We do not consider the property as failing if eventualities are not satisfied by the end of the simulation. Doing so would require changing the semantics of the specification and would require special treatment of the last state. Our approach maintains the same semantics for dynamic and formal verification runs and only bad prefixes are reported as failures.

The workflows that we use to generate monitors can be grouped into two types, summarized in Fig. 1.

5 Monitor generation

We now describe various issues that arise when constructing $\mathcal{A}_{DFW}(\psi)$.

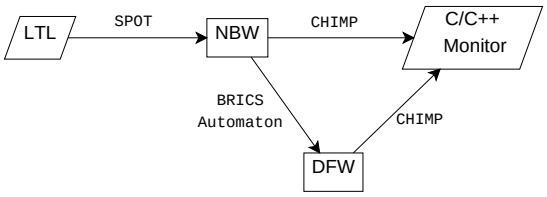

Fig. 1 The two types of workflows we used to generate monitors. The focus of our work is on the paths from NBW to a monitor; we use SPOT as a pre-processor to generate pruned NBWs from LTL formulas.

5.1 State minimization

As noted above, we can construct $\mathcal{A}_{DFW}(\psi)$ on the fly. We discuss in detail below how to express $\mathcal{A}_{DFW}(\psi)$ as a collection of `C++` expressions. The alternative is to feed $\mathcal{A}_{NFW}(\psi)$ into a tool that constructs a minimal equivalent $\mathcal{A}_{DFW}(\psi)$. We use the `BRICS Automaton` tool [34]. Clearly, determinization and minimization, as well as subsequent `C++` compilation, may incur a non-trivial computational cost. Still such a cost might be justifiable if the result is reduced *runtime* overhead, as assertions have to be compiled only once, but then run many times. A key question we want to answer is whether it is worthwhile to determinize $\mathcal{A}_{NFW}(\psi)$ explicitly, rather than on the fly.

5.2 Alphabet representation

In our formalism, the alphabet Σ of $\mathcal{A}_{NFW}(\psi)$ is $\Sigma = 2^{AP}$, where AP is the set of atomic propositions appearing in ψ. In practice, tools that generate $\mathcal{A}_{NBW}(\psi)$ (SPOT in our case) often use $\mathcal{B}(AP)$, the set of Boolean formulas over AP, as the automaton alphabet: a transition from state q to state q' labeled by the formula θ is a shortcut to denote all transitions from q to q' labeled by $\sigma \in 2^{AP}$, when σ satisfies θ. When constructing $\mathcal{A}_{DFW}(\psi)$ on the fly, we can use formulas as letters. Automata-theoretic algorithms for determinization and minimization of NFWs, however, require comparing elements of Σ, which makes it impractical to use Boolean formulas for letters. We need a different way, therefore, to describe our alphabet.[5] Below we show two ways to describe the alphabet of $\mathcal{A}_{NFW}(\psi)$ in terms of 16-bit integers.

5.2.1 Assignment-based representation

The explicit approach is to represent Boolean formulas in terms of their satisfying truth *assignments*. Let $AP = \{p_1, p_2, \ldots, p_n\}$ and let $\mathcal{F}(p_1, p_2, \ldots, p_n)$ be a Boolean function. An *assignment* to AP is an n-bit vector $\mathbf{a} = [a_1, a_2, \ldots, a_n]$. An assignment $\mathbf{a}$ *satisfies* $\mathcal{F}$ iff $\mathcal{F}(a_1, a_2, \ldots, a_n)$ evaluates to 1. Let A^n be

[5] `BRICS Automaton` represents the alphabet of the automaton as Unicode characters, which have one-to-one correspondence to the set of 16-bit integers.

the set of all n-bit vectors and let $I : A^n \to \mathbb{Z}_+$ return the integer whose binary representation is $\mathbf{a}$, i.e., $I(a) = a_1 2^{n-1} + a_2 2^{n-2} + \ldots + a_n 2^0$. We define $sat(\mathcal{F}) = \{I(a) : a \text{ satisfies } \mathcal{F}\}$. Thus, the explicit representation of the automaton $\mathcal{A}_{NFW}(\psi) = \langle \mathcal{B}(AP), Q, \delta, Q^0, F\rangle$ is $\mathcal{A}^{abr}_{NFW}(\psi) = \langle \{0, \ldots, 2^n - 1\}, Q, \delta_{abr}, Q^0, F\rangle$, where $q' \in \delta_{abr}(q, z)$ iff $q' \in \delta(q, \sigma)$ and $z \in sat(\sigma)$.

5.2.2 BDD-based representation

The symbolic approach to alphabet representation leverages the fact that Ordered Binary Decision Diagrams (*BDD*s) [6, 7] provide canonical representations of Boolean functions. A BDD is a rooted, directed, acyclic graph with one or two terminal nodes labeled **0** or **1**, and a set of variable nodes of outdegree two. The variables respect a given linear order on all paths from the root to a leaf. Each path represents an assignment to each of the variables on the path. For a fixed variable order, two BDDs are the same iff the Boolean formulas they represent are the same.

The symbolic approach uses SPOT's `spot::tgba_reachable_iterator_breadth_first::process_link()` function call to get references to all Boolean formulas that appear as transition labels in $\mathcal{A}_{NFW}(\psi)$. The formulas are enumerated using their BDD representations (using SPOT's `spot::tgba_succ_iterator::current_condition()` function call), and each unique formula is assigned a unique integer. We thus obtain $\mathcal{A}^{bdd}_{NFW}(\psi)$ by replacing transitions labeled by Boolean formulas with transitions labeled by the corresponding integers. While the size of $\mathcal{B}(AP)$ is doubly exponential in $|AP|$, the automaton $\mathcal{A}_{NBW}(\psi)$ is exponential in $|\psi|$, so the number of Boolean formulas used in the automaton is at most exponential in $|\psi|$.

5.2.3 From NFW to DFW

We provide both $\mathcal{A}^{abr}_{NFW}(\psi)$ and $\mathcal{A}^{bdd}_{NFW}(\psi)$ as inputs to `BRICS Automaton`, producing, respectively, minimized $\mathcal{A}^{abr}_{DFW}(\psi)$ and $\mathcal{A}^{bdd}_{DFW}(\psi)$. We note that neither of these two approaches is a priori a better choice. LTL–to–automata tools use Boolean formulas rather than assignments to reduce the number of transitions in the generated *nondeterministic* automata. However, when using $\mathcal{A}^{bdd}_{DFW}(\psi)$ as a monitor, the trace we monitor is a sequence of truth assignments, and $\mathcal{A}^{bdd}_{DFW}(\psi)$, while deterministic with respect to the BDD encoding of the transitions, is not deterministic with respect to truth assignments to atomic propositions. As a consequence, there is no guarantee that at each step of the monitor at most one state is reachable.

5.3 Alphabet minimization

While propositional temporal specification languages are based on Boolean atomic propositions, they are often used to specify properties involving non-Boolean variables. For example, we may have the atomic formulas `(a == 0)`,

`(a == 1)`, and `(a > 1)` in a specification involving the values of a variable **`int`** `a`. Notice that in this example not all assignments in 2^{AP} are consistent. For example, the assignment `(a == 0) && (a == 1)` is not consistent, and a transition guarded by `(a == 0) && (a == 1)` is never enabled. Note that such a guard can be generated even if the guard is not a subformula in the specification. By eliminating inconsistent assignments we may be able to reduce the number of letters in the alphabet exponentially without in any way changing the correctness of the monitor. The advantage of this optimization is that by identifying transitions that always evaluate to ***false*** we can exclude them from the generated monitor and thus improve its run-time performance. Identifying inconsistent assignments requires calling an SMT (Satisfiability Modulo Theory) solver [36]. Here we would need an SMT solver that can handle arbitrary `C++` expressions that evaluate to type **`bool`**. Not having access to such an SMT solver, we use the compiler as an improvised SMT solver.

A set of techniques called *constant folding* allows compilers to reduce constant expressions to a single value at compile time (see, e.g., [12]). When an expression contains variables instead of constants, the compiler uses *constant propagation* to substitute values of variables in subsequent subexpressions involving the variables. In some cases the compiler is able to deduce that an expression contains two mutually exclusive subexpressions, and issues a warning during compilation. We construct a function that uses conjunctions of atomic formulas as conditionals for dummy `if/then` expressions, and compile the function. (We use `gcc 4.0.3`.) To gauge the effectiveness of this optimization we apply it using two sets of conjunctions. *Full alphabet minimization* uses all possible conjunctions involving atomic formulas or their negations, while *partial alphabet minimization* uses only conjunctions that contain each atomic formula, positively or negatively.

We compile the function and then parse the compiler warnings that identify inconsistent conjunctions. Prior to compiling the Büchi automaton we augment the original temporal formula to exclude those conjunctions from consideration. For example, if `(a == 0) && (a == 1)` is identified as an inconsistent conjunction, we augment the property ψ to $\psi \wedge \mathbf{G}(!((a == 0) \wedge (a == 1)))$.

5.4 Monitor encoding

We describe seven ways of encoding automata as `C++` monitors. Not all can be used with all automata directly, so we identify the transformations that need to be applied to an automaton before each encoding can be used.

The strategy in all encodings based on automata that are nondeterministic with respect to truth assignments (i.e., $\mathcal{A}_{NFW}(\psi)$ and minimal $\mathcal{A}^{bdd}_{DFW}(\psi)$) is to construct the run $P_0, P_1, \ldots$ of the monitor using two bit-vectors of size $|Q|$: `current[]` and `next[]`. Initially `next[]` is zeroed, and `current[j] = 1` iff $q_j \in Q^0$. Then, after sampling the state of the program, we set `next[k] = 1` iff `current[j] = 1` and if there is a transition from q_j to q_k that is enabled

by the current program state. When we are done updating `next[]`, we assign it to `current[]`, zero `next[]`, and then repeat the process at the next sample point. Intuitively, `current[]` keeps track of the set of automaton states that are reachable after seeing the execution trace so far, and `next[]` maintains the set of automaton states that are reachable after completing the current step of the automaton.

Notice that when the underlying automaton is deterministic with respect to truth assignments (i.e., $\mathcal{A}^{abr}_{DFW}(\psi)$), after each step there are precisely 1 or 0 reachable states. In those cases it is inefficient to use bit-vector encoding of the set of reachable states, because this set is guaranteed to be singleton. Thus, when constructing monitors from deterministic automata, we use **`int`** `current` and **`int`** `next` to keep track of the run of the automaton. Initially, `current = j` iff q_j is the initial state. Then we set `next = k` iff the transition from q_j to q_k is enabled at the first sample point; since the automaton is deterministic, at most one transition is enabled. We continue in this fashion until the simulation ends or until none of the transitions in the monitor is enabled, indicating a bad prefix.

The details of the way we update `current[]` (respectively, `current`) and `next[]` (respectively, `next`) are reflected in the different encodings. As a running example, we show how to construct a monitor for the property $\varphi = \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}q \wedge \mathbf{XX}q))$. The first step is to use SPOT to construct a NBW automaton that accepts all traces satisfying φ. Next, we use SPOT to construct $\mathcal{A}_{NFW}(\varphi)$, which is presented in Figure 2.

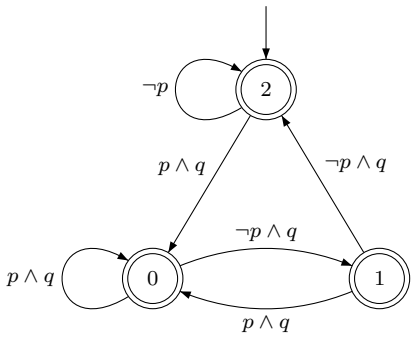

Fig. 2 $\mathcal{A}_{NFW}(\varphi)$ constructed from the specification $\varphi = \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}q \wedge \mathbf{XX}q))$ using the algorithm of d'Amorim and Roşu. Double circles represent accepting states, and state 2 is the initial state.

5.4.1 Nondeterministic encodings

Two novel encodings, which we call `front_nondet` and `back_nondet`, expect that the automaton transitions are Boolean formulas, and do not as-

sume determinism. Thus, `front_nondet` and `back_nondet` can be used with $\mathcal{A}_{NFW}(\psi)$ directly. They can also be used with $\mathcal{A}^{abr}_{DFW}(\psi)$ and $\mathcal{A}^{bdd}_{DFW}(\psi)$, once we convert back the transition labels from integers to Boolean formulas as follows. In $\mathcal{A}^{abr}_{DFW}(\psi)$, we calculate the assignment corresponding to each integer, and use that assignment to generate a conjunction of atomic formulas or their negations. In $\mathcal{A}^{bdd}_{DFW}(\psi)$ we relabel each transition with the Boolean function whose BDD is represented by the integer label.

The `front_nondet` encoding uses an explicit **`if`** to check if each state s of `current[]` is enabled. For each outgoing transition t from s it then uses a nested **`if`** with a conditional that is a verbatim copy of the transition label of t to determine if the destination state of t is reachable from s. Listing 3 illustrates this encoding.

```
1  /**
2   * front_nondet encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
6      // Property has not been determined to fail so far
7      num_steps++; // Current length of execution trace
8
9      for (int i = 0; i < 3; i++) { //Loop through all of the states
10       current[i] = next[i];       // saving the next state and
11       next[i] = 0;                // clearing the next state vector
12     }
13
14       if (current[0]) { //if the current state is state 0
15         if(!(p) && (q)) //if transition labeled !(p)&&(q) is enabled
16           next[1] = 1;  //state 1 is a possible next state
17         if((p) && (q))  //if transition labeled (p)&&(q) is enabled
18           next[0] = 1;  //state 0 is a possible next state
19       } // if
20
21       if (current[1]) { //if the current state is state 1
22         if((p) && (q))  //if transition labeled (p)&&(q) is enabled
23           next[0] = 1;  //state 0 is a possible next state
24         if(!(p) && (q)) //if transition labeled !(p)&&(q) is enabled
25           next[2] = 1;  //state 2 is a possible next state
26       } // if
27
28       if (current[2]) {  //if the current state is state 2
29         if((p) && (q))   //if transition labeled (p)&&(q) is enabled
30           next[0] = 1;   //state 0 is a possible next state
31         if(!(p) && (q))  //if transition labeled !(p)&&(q) is enabled
32           next[2] = 1;   //state 2 is a possible next state
33         if(!(p) && !(q)) //if transition labeled !(p)&&!(q) is enabled
34           next[2] = 1;   //state 2 is a possible next state
35       } // if
36
37     // Check if there were enabled transitions
38     bool not_stuck = false;
39     for (int i = 0; i < 3; i++) { //Loop through all of the states
40       not_stuck = not_stuck || next[i];
41     }
42
```

```
43      if (! not_stuck) {
44        // None of the transitions were enabled
45
46  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
47        SC_REPORT_WARNING("Property failed", "Critical error");
48        std::cout << "Property failed after " << num_steps
49                  << " steps" << std::endl;
50  #endif
51        status = MON_FAIL; //flag the specification failure
52      }
53    } // if (status == MON_UNDETERMINED)
54  } // step()
```

Listing 3 Illustrating `front_nondet` encoding of the automaton in Figure 2.

The `back_nondet` encoding uses a disjunction that represents all of the ways in which a state in `next[]` can be reached from the currently reachable states. Listing 4 illustrates this encoding.

```
1   /**
2    * back_nondet encoding
3    */
4   test_monitor0::step() {
5     if (status == MON_UNDETERMINED) {
6       // Property has not been determined to fail so far
7       num_steps++; // Current length of execution trace
8
9       for (int i = 0; i < 3; i++) { //Loop through all of the states
10        current[i] = next[i];       // saving the next state and
11        next[i] = 0;                // clearing the next state vector
12      }
13
14      //Determine which states are enabled next time
15      // based on the current state and the enabled transition(s)
16      next[0] = ( current[2] && ((p) && (q))) ||
17                ( current[1] && ((p) && (q))) ||
18                ( current[0] && ((p) && (q)));
19
20      next[1] = ( current[0] && (!(p) && (q)));
21
22      next[2] = ( current[2] && (!(p) && (q))) ||
23                ( current[1] && (!(p) && (q))) ||
24                ( current[2] && (!(p) && !(q)));
25
26      // Check if there were enabled transitions
27      bool not_stuck = false;
28      for (int i = 0; i < 3; i++) { //Loop through all of the states
29        not_stuck = not_stuck || next[i];
30      }
31
32      if (! not_stuck) {
33        // None of the transitions were enabled
34
35  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
36        SC_REPORT_WARNING("Property failed", "Critical error");
37        std::cout << "Property failed after " << num_steps
38                  << " steps" << std::endl;
39  #endif
```

```
40        status = MON_FAIL; //flag the specification failure
41      }
42    } // if (status == MON_UNDETERMINED)
43  } // step()
```

Listing 4 Illustrating `back_nondet` encoding of the automaton in Figure 2.

5.4.2 Deterministic encodings

Three novel deterministic encodings, which we call `front_det_switch`, `front_det_ifelse`, and `back_det`, expect that the automaton has been determinized using assignment-based encoding. Thus, these three encodings can be used only with $\mathcal{A}^{abr}_{DFW}(\psi)$. Note that we work with $\mathcal{A}^{abr}_{DFW}(\psi)$ directly and do not convert the automaton alphabet from integers back to Boolean functions. Instead, at the beginning of each step of the automaton we use the state of the MUV (i.e., the values of all public and private variables, as exposed by the framework of [42]) to derive an assignment $\mathbf{a}$ to the atomic propositions in $AP(\psi)$. We then calculate an integer representing the relevant model state $mod\_st = I(\mathbf{a})$, where $\mathbf{a}$ is the current assignment, and use mod_st to drive the automaton transitions.

Referring to the running example automaton presented in Figure 2, we first show how to convert the Boolean expressions on the transitions to integers using assignment-based integer representation. Table 1 shows the integer encoding of all possible assignments of values to p and q. We then construct $\mathcal{A}^{abr}_{NFW}(\varphi)$ in Figure 3. Determinizing and minimizing $\mathcal{A}^{abr}_{NFW}(\varphi)$ using `BRICS Automaton` produces $\mathcal{A}^{abr}_{DFW}(\varphi)$, which in this case is identical to $\mathcal{A}^{abr}_{NFW}(\varphi)$.

p	q	`int`
0	0	0
0	1	1
1	0	2
1	1	3

Table 1 Assignment-based encoding for the transitions of the $\mathcal{A}_{NFW}(\psi)$ in Figure 2.

The `back_det` encoding is similar to `back_nondet` in that it encodes the automaton transitions as a disjunction of the conditions that allow a state in `next[]` to be enabled. The difference is that here we use an integer instead of a vector to keep track of the (at most one) state reachable in the current step of the automaton, and the transitions are driven by mod_st instead of by Boolean functions. See Listing 5 for an illustration of this encoding.

```
1  /**
2   * back_det encoding
3   */
4  test_monitor0::step() {
5    if (status == MON_UNDETERMINED) {
```

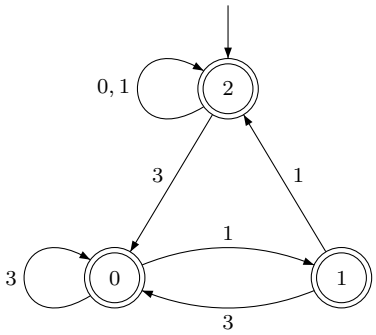

Fig. 3 $\mathcal{A}^{abr}_{NFW}(\varphi)$ for $\varphi = \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}q \wedge \mathbf{XX}q))$. Determinizing $\mathcal{A}^{abr}_{NFW}(\varphi)$ using `BRICS Automaton` produces $\mathcal{A}^{abr}_{DFW}(\varphi)$, which is then minimized. The minimized $\mathcal{A}^{abr}_{DFW}(\varphi)$ in this case is identical to $\mathcal{A}^{abr}_{NFW}(\varphi)$.

```
6       // Property has not been determined to fail so far
7       num_steps++; // Current length of execution trace
8
9       //shift the next state into the current state position
10      current = next;
11
12      next = -1; //clear the next state
13
14      // Calculate the system state index
15      // using the enabled alphabet characters
16      int system_state_index = 0;
17      system_state_index += (p) ? (1 << 1) : 0;
18      system_state_index += (q) ? (1 << 0) : 0;
19
20      //Determine which state is enabled next time
21      // based on the current state and the enabled transition
22      if (( ( current == 2) && ( system_state_index == 3)) ||
23          ( ( current == 1) && ( system_state_index == 3)) ||
24          ( ( current == 0) && ( system_state_index == 3)))
25         { next = 0;}
26      else if (( ( current == 0) && ( system_state_index == 1)))
27         { next = 1;}
28      else if (( ( current == 2) && ( system_state_index == 1)) ||
29               ( ( current == 1) && ( system_state_index == 1)) ||
30               ( ( current == 2) && ( system_state_index == 0)))
31         { next = 2;}
32
33      // Check if there were enabled transitions
34      bool not_stuck = (next != -1);
35      if (! not_stuck) {
36        // None of the transitions were enabled
37
38  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
39        SC_REPORT_WARNING("Property failed", "Critical error");
40        std::cout << "Property failed after " << num_steps
41                  << " steps" << std::endl;
```

```
#endif
      status = MON_FAIL; //flag the specification failure
    }
  } // if (status == MON_UNDETERMINED)
} // step()
```

Listing 5 Illustrating `back_det` encoding of the automaton in Figure 3.

The `front_det_switch` and `front_det_ifelse` encodings are similar, but differ in the `C++` constructs used to take advantage of the determinism in the automaton. Applying `front_det_switch` encoding to the automaton in Figure 3 is illustrated in Listing 6, and `front_det_ifelse` encoding is illustrated in Listing 7.

```
/**
 * front_det_switch encoding
 */
test_monitor0::step() {
  if (status == MON_UNDETERMINED) {
    // Property has not been determined to fail so far
    num_steps++; // Current length of execution trace

    //shift the next state into the current state position
    current = next;

    next = -1; //clear the next state

    // Calculate the system state index
    // using the enabled alphabet characters
    int system_state_index = 0;
    system_state_index += (p) ? (1 << 1) : 0;
    system_state_index += (q) ? (1 << 0) : 0;

    //Determine which state is enabled next time
    // based on the current state and the enabled transition
    switch (current) {
      case 0: //if the current state is state 0...
        switch ( system_state_index ) {
          case 1: next = 1; break;
          case 3: next = 0; break;
        } // inner switch/case
      break; // the outer case

      case 1: //if the current state is state 1...
        switch ( system_state_index ) {
          case 3: next = 0; break;
          case 1: next = 2; break;
        } // inner switch/case
      break; // the outer case

      case 2: //if the current state is state 2...
        switch ( system_state_index ) {
          case 3: next = 0; break;
          case 1: next = 2; break;
          case 0: next = 2; break;
        } // inner switch/case
      break; // the outer switch/case
```

```
    } // switch (current)

    // Check if there were enabled transitions
    bool not_stuck = (next != -1);
    if (! not_stuck) {
      // None of the transitions were enabled

#ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
      SC_REPORT_WARNING("Property failed", "Critical error");
      std::cout << "Property failed after " << num_steps
                << " steps" << std::endl;
#endif
      status = MON_FAIL; //flag the specification failure
    }
  } // if (status == MON_UNDETERMINED)
}
```

Listing 6 Illustrating `front_det_switch` encoding of the automaton in Figure 3.

```
/**
 * front_det_ifelse encoding
 */
test_monitor0::step() {
  if (status == MON_UNDETERMINED) {
    // Property has not been determined to fail so far
    num_steps++; // Current length of execution trace

    //shift the next state into the current state position
    current = next;

    next = -1; //clear the next state

    // Calculate the system state index
    // using the enabled alphabet characters
    int system_state_index = 0;
    system_state_index += (p) ? (1 << 1) : 0;
    system_state_index += (q) ? (1 << 0) : 0;

    if (current == 0) { //if the current state is state 0...
      if ( system_state_index == 1 ) { next = 1; }
      else if ( system_state_index == 3 ) { next = 2; }
    } // if (current == ...)

    else if (current == 1) { //if the current state is state 1...
      if ( system_state_index == 1 ) { next = 1; }
      else if ( system_state_index == 3 ) { next = 2; }
      else if ( system_state_index == 0 ) { next = 1; }
    } // if (current == ...)

    else if (current == 2) { //if the current state is state 2...
      if ( system_state_index == 3 ) { next = 2; }
      else if ( system_state_index == 1 ) { next = 0; }
    } // if (current == ...)

    // Check if there were enabled transitions
    bool not_stuck = (next != -1);
    if (! not_stuck) {
      // None of the transitions were enabled
```

```
40
41  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
42        SC_REPORT_WARNING("Property failed", "Critical error");
43        std::cout << "Property failed after " << num_steps
44                  << " steps" << std::endl;
45  #endif
46        status = MON_FAIL; //flag the specification failure
47      }
48    }
49  }
```

Listing 7 Illustrating `front_det_ifelse` encoding of the automaton in Figure 3.

5.5 Deterministic table-based encodings

In the encodings discussed above, the transition function of the automaton is encoded using `if` or `switch` statements. The final two encodings, described below, are based on table look-up. The key to a table look-up monitor encoding is to create a table, such that given the current state and the current assignment, we can look up the next state in the table. Given the current state and the system state index *mod_st*, we can transition to the next state in one operation, avoiding overhead associated with large nested `if` statements or `switch` statements.

We illustrate both table-based encodings using the determinized, minimized automaton $\mathcal{A}^{abr}_{DFW}(\varphi)$ presented in Figure 3 and the associated integer encodings of all possible assignments of values to p and q in Table 1. We can construct a look-up table as illustrated in Table 2.

		alphabet representation			
		0	1	2	3
current state	0	fail	1	fail	0
	1	fail	2	fail	0
	2	2	2	fail	0

Table 2 Look-up table corresponding to the automaton in Fig. 3. Given the current state and the integer representation of the alphabet, find the next state or detect failure. For example, if the current state is 0 and the alphabet representation is 3, the next state is state 0.

Two novel deterministic encodings, which we call `front_det_file_table` and `front_det_memory_table`, expect that the automaton has been determinized using assignment-based encoding. Like the other deterministic encodings, these two encodings can be used only with $\mathcal{A}^{abr}_{DFW}(\psi)$. Again, we work with $\mathcal{A}^{abr}_{DFW}(\psi)$ directly and do not convert the automaton alphabet from integers back to Boolean functions; for table encodings we take advantage of the fact that the automaton alphabet integers can be stored easily in a state-transition look-up table.

As we did for the encodings of Section 5.4.2, we use the state of the MUV (i.e., the values of all public and private variables, as exposed by the framework of [42]) at the beginning of each step of the automaton to derive an assignment $\mathbf{a}$ to the atomic propositions in $AP(\psi)$. Again, we calculate an integer representing the relevant model state $mod\_st = I(\mathbf{a})$, where $\mathbf{a}$ is the current assignment.

The encodings `front_det_file_table` and `front_det_memory_table` are similar, but differ in the way the state-transition look-up table is stored and used by the monitor.

5.5.1 The file-based table encoding

In the `front_det_file_table` encoding we store the automaton $\mathcal{A}^{abr}_{DFW}(\varphi)$ in a text file using the LBT format [38]. Briefly, the LBT file format is a text-based encoding of automata. It iteratively describes each state (whether it is accepting, initial, both or neither), together with a unique state ID. Each outgoing transition is listed immediately after the state description, and includes the destination state ID and a transition letter/guard.

When the monitor is instantiated, it uses an LBT parser that is automatically included with the monitor's code to parse the automaton from the file and to construct the look-up table. Applying `front_det_file_table` encoding to the automaton in Figure 3 is illustrated in Listing 8.

```
1  /**
2   * front_det_file_table encoding: constructor
3   */
4  test_monitor::test_monitor() {
5    next = 2; // initial state id
6    current = -1;
7    //other settings go here...
8
9    // LBT encoding of the automaton implemented by this monitor
10   const char* automaton_file = "./A_DFW.lbt";
11
12   int size_of_alphabet = 4;
13
14   // Variable "table", below, is anum_states x size_of_alphabet 2-D
15   // array declared in the header as a class variable
16   table = (new lbt_parser())->parse_to_table(automaton_file,
17                                              size_of_alphabet);
18 } // end constructor
19
20 /**
21 * front_det_file_table encoding: simulate a step of the monitor.
22 */
23 test_monitor::step() {
24   if (status == MON_UNDETERMINED) {
25     // Property has not been determined to fail so far
26     num_steps++; // Current length of execution trace
27
28     // Shift the next state into the current state position
29     current = next;
30
```

```
31      // Calculate the system state index
32      // using the enabled alphabet characters
33      int system_state_index = 0;
34      system_state_index += (p) ? (1 << 1) : 0;
35      system_state_index += (q) ? (1 << 0) : 0;
36
37      // The table is a 2-d array declared in the constructor
38      // Look up the next state
39      next = table[current][system_state_index];
40
41      // Check if there were enabled transitions
42      bool not_stuck = (next != -1);
43      if (! not_stuck) {
44        // None of the transitions were enabled
45
46  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
47        SC_REPORT_WARNING("Property failed", "Critical error");
48        std::cout << "Property failed after " << num_steps
49                  << " steps" << std::endl;
50  #endif
51        status = MON_FAIL; //flag the specification failure
52      }
53    } // if (status == MON_UNDETERMINED)
54  } //end step
```

Listing 8 Illustrating `front_det_file_table` encoding of the automaton in Figure 3.

One advantage of using the file-based table encoding is that it separates the code implementing the monitor from the definition of the automaton. Such decoupling allows the monitor to be compiled and linked with the MUV before the LBT representation of the automaton is even created. It further allows monitored properties to be changed on the fly, without having to recompile the MUV, by simply replacing the contents of the LBT file.

5.5.2 The memory table encoding

In the `front_det_memory_table` encoding we declare the state-transition look-up table explicitly in the monitor's constructor. The table is declared directly as a one-dimensional, row-major array, forgoing the need for a LBT parser library or a file containing the automaton. Applying the `front_det_memory_table` encoding to the automaton in Figure 3 is illustrated in Listing 9.

```
1   /**
2    * front_det_memory_table encoding: constructor
3    */
4   test_monitor::test_monitor()  {
5     next = 0; // initial state id
6     current = -1;
7     //other settings go here...
8
9     //Explicitly declare the transition table as a row-major 1-D array
10    int local_table[] = {-1, 1, -1, 0, -1, 2, -1, 0, 2, 2, -1, 0};
11
12    // Variable "table", below, is a num_states x size_of_alphabet 1-D
13    // array declared in the header as a class variable. We memcpy
```

```
14    // local_table to table to avoid a limitation of C++.
15    std::memcpy(table, local_table, 12 * sizeof(int));
16  } // end constructor
17
18  /**
19  * front_det_memory_table encoding: simulate a step of the monitor.
20  */
21  test_monitor0::step() {
22    if (status == MON_UNDETERMINED) {
23      // Property has not been determined to fail so far
24      num_steps++; // Current length of execution trace
25
26      //shift the next state into the current state position
27      current = next;
28
29      // Calculate the system state index
30      // using the enabled alphabet characters
31      unsigned int system_state_index = 0;
32      system_state_index += (p) ? (1 << 1) : 0;
33      system_state_index += (q) ? (1 << 0) : 0;
34
35      // Look up the next state in the table
36      next = table[current * 4 + system_state_index];
37
38      // Check if there were enabled transitions
39      bool not_stuck = (status == MON_PASS) || (next != -1);
40      if (! not_stuck) {
41        // None of the transitions were enabled
42
43  #ifdef MONITOR_REPORT_FAIL_IMMEDIATELY
44        SC_REPORT_WARNING("Property failed", "Critical error");
45        std::cout << "Property failed after " << num_steps
46                  << " steps" << std::endl;
47  #endif
48        status = MON_FAIL; //flag the specification failure
49      }
50    } // if (status == MON_UNDETERMINED)
51  } //end step
```

Listing 9 Illustrating `front_det_memory_table` encoding of the automaton in Figure 3.

Note that in this encoding the variable `table` is a class variable of type `int[]` with the same capacity as the number of elements in `local_table`. A limitation of the current `C++` standard (C99) does not allow arrays to be initialized explicitly after declaration, thus we use `local_table` to initialize the array, and `std::memcpy` to copy the array from `local_table` to `table`.

5.6 Workflow space

The different options allow 33 possible combinations for generating a monitor, summarized in Table 3. The first decision is whether state minimization is required. If it is not required, one of the three alphabet minimization options is applied, and one of the two non-deterministic monitor encodings (`front_nondet` or `back_nondet`) is used to create the final monitor.

When using state minimization it is necessary to select the alphabet representation (BDD- or assignment-based) to be using during minimization. The three alphabet minimization options can be selected independently of the alphabet representation selection. Recall that BDD-based minimization produces automata that are non-deterministic with respect to assignments, therefore only the two non-deterministic monitor encodings are available. Alternatively, if assignment-based minimization is employed, all non-deterministic and deterministic encodings (seven total) can be used.

In summary, there are six workflows that require no state minimization, six workflows that use BDD-based state minimization, and 21 workflows that use assignment-based state minimization.

State Minimization	Alphabet Representation	Alphabet Minimization	Monitor Encoding
no	Not required	none	`front_nondet`
			`back_nondet`
yes	BDDs		`front_nondet`
			`back_nondet`
	assignments	none	`front_nondet`
			`back_nondet`
		partial	`front_det_ifelse`
			`front_det_switch`
		full	`back_det`
			`front_det_file_table`
			`front_det_memory_table`

Table 3 The workflow space for generating monitors.

6 Experimental setup

6.1 SystemC model

Our experimental evaluation is based on the Adder[6] model presented in [41]. The Adder implements a squaring function by using repeated incrementing by 1. We used the Adder to calculate 100^2 with 1,000 instances of a monitor for the same property. Since we are mostly concerned with monitor overhead, we focus on the time difference between executing the model with and without monitoring. We established a baseline for the model's runtime by compiling the Adder model with a virgin installation of SystemC (i.e., without the monitoring framework of [41]) and averaging the runtime over 10 executions. To

[6] Source code available at `http://www.cs.rice.edu/CS/Verification/Software/software.html`

calculate the monitor overhead we averaged the runtime of each simulation over 10 executions and subtracted the baseline time. Notice that the overhead as calculated includes the cost of the monitoring framework and the slow-down due to all 1,000 monitors.

6.2 Properties

We used specifications constructed using both pattern formulas and randomly generated formulas. We used LTL formulas, as we have access to explicit-state LTL-to-automata translators (SPOT, in our case). Note, however, that the framework is applicable to any specification language that produces NBWs and is not restricted to LTL formulas. Minimization of finite-state automata was performed by `BRICS Automaton`. SPOT, `BRICS Automaton`, and CHIMP, the tool that manages the different workflows, are available for download[7].

We adopted the pattern formulas used in [21] and presented below:

$$c1(n) := \bigvee_{i=1}^{n} \mathbf{GF}p_i \qquad lu(n) := (\ldots(p_1\mathbf{U}p_2))\ldots\mathbf{U}p_n)\mathbf{U}p_{n+1}$$

$$c2(n) := \bigwedge_{i=1}^{n} \mathbf{GF}p_i \qquad ru(n) := p_1\mathbf{U}(p_2\mathbf{U}(\ldots(p_n\mathbf{U}p_{n+1})\ldots))$$

$$qq(n) := \bigwedge_{i=1}^{n} (\mathbf{F}p_i \vee \mathbf{G}p_{i+1}) \qquad rr(n) := \bigwedge_{i=1}^{n} (\mathbf{GF}p_i \vee \mathbf{FG}p_{i+1})$$

$$ss(n) := \bigvee_{i=1}^{n} \mathbf{G}p_i$$

In addition to these formulas we also used bounded **F** and bounded **G** formulas, and a new type of nested **U** formulas, presented below:

$$f1(n) := \mathbf{G}(p \rightarrow (q \vee \mathbf{X}q \vee \ldots \vee \mathbf{XX}\ldots\mathbf{X}q))$$
$$f2(n) := \mathbf{G}(p \rightarrow (q \vee \mathbf{X}(q \vee \mathbf{X}(q \vee \ldots \vee \mathbf{X}q)\ldots)))$$
$$g1(n) := \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}q \wedge \ldots \wedge \mathbf{XX}\ldots\mathbf{X}q))$$
$$g2(n) := \mathbf{G}(p \rightarrow (q \wedge \mathbf{X}(q \wedge \mathbf{X}(q \wedge \ldots \wedge \mathbf{X}q)\ldots)))$$
$$uu(n) := \mathbf{G}(p_1 \rightarrow (p_1\mathbf{U}(p_2 \wedge p_2\mathbf{U}(p_3\ldots(p_n \wedge p_n\mathbf{U}p_{n+1}))))\ldots)$$

In our experiments we replaced the generic propositions p_i in each pattern formula with atomic formulas `(a==100^2-100(n-i-1))`, where `a` is a variable representing the running total in the Adder. For each pattern we scaled up the formulas until all 33 workflows either timed out or crashed. Most workflows can be scaled up to $n = 5$, except for the bounded properties, which can be scaled to $n = 17$. We identified 127 pattern formulas for which at least one workflow could complete the monitoring task.

[7] `http://www.cs.rice.edu/CS/Verification/Software/software.html`

The random formulas that we used were generated following the framework of [15], using the implementation from [39]. For each formula length there are two parameters that control the number of propositions used and the probability of selecting a **U** or a **V** operator (formula length is calculated by adding the number of atomic propositions, the number of logical connectives, and the number of temporal operators). We varied the number of atomic propositions between 1 and 5, the probability of selecting a **U** or a **V** was one of {0.3, 0.5, 0.7, 0.95}, and we varied the formula length from 5 to 30 in increments of 5. We used the same style of atomic propositions as in the pattern formulas. For each combination of parameters we generated 10 formulas at random, giving us a total of 1200 random formulas.

7 Results for non-table-based workflows

The results described in this section are based on experiments on Ada, Rice's Cray XD1 compute cluster.[8] Each of Ada's nodes has two dual core 2.2 GHz AMD Opteron 275 CPUs and 8GB of RAM. We ran with exclusive access to a node so all 8GB of RAM were available for use. We allowed 8 hours (the maximal job time on Ada) of computation time per workflow per formula for generating a Büchi automaton, automata-theoretic transformations, generating `C++` code, compilation, linking with the Adder model using the monitoring framework presented in [41], and executing the monitored model 10 times.

We first evaluate the individual effect of each optimization. For each formula we partition the workflow space into two groups: those workflows that use the optimization and those that do not. We form the Cartesian product of the overhead times from both groups and present them on a scatter plot.

7.1 State minimization

Fig. 4 shows the effect of determinization and state minimization on the automaton size. We observe that in most cases minimizing the automata (i.e., minimizing $\mathcal{A}^{abr}_{DFW}(\varphi)$ and $\mathcal{A}^{bdd}_{DFW}(\varphi)$) produces smaller automata than the equivalent $\mathcal{A}_{NFW}(\varphi)$. It is known [28] that in the worst case, nondeterministic automata are exponentially more succinct than the corresponding minimal deterministic automata. Our experimental results show that the worst case blow up is avoided for the types of formulas that are likely to be used in practice, and, in fact, for some formulas we see three orders of magnitude smaller deterministic automata. This observation goes against the traditional justification for constructing monitors from nondeterministic rather than deterministic automata.

In Fig. 5 we show the effect of state minimization on the runtime overhead. A few outliers notwithstanding, using state minimization lowers the runtime overhead of the monitor.

[8] `http://www.rcsg.rice.edu/ada`

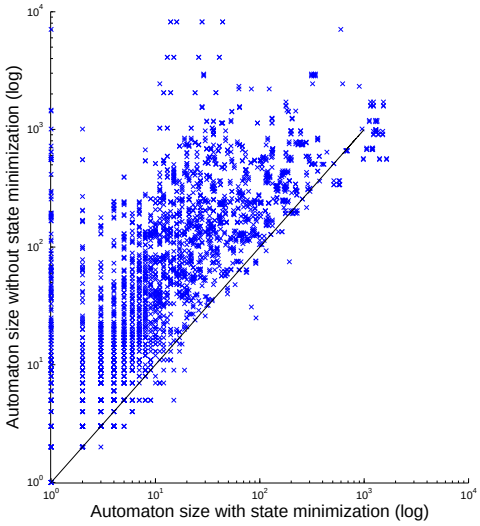

Fig. 4 The size of the determinized/minimized automaton in most cases is smaller than the size of the corresponding nondeterministic automaton. Points fall above the diagonal when this is the case.

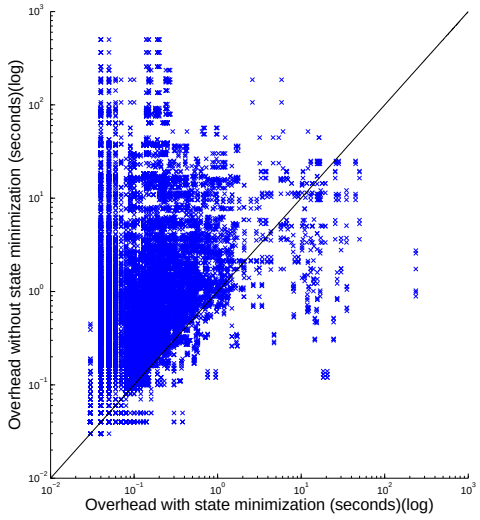

Fig. 5 Monitor overhead with and without state minimization. State minimization lowers the overhead by orders of magnitude. Points fall above the diagonal when monitor overhead with state minimization is lower.

7.2 Alphabet Representation

Fig. 6 shows that using assignments leads to better performance than BDD-based alphabet representation. Our data show that in most cases, using assign-

ments leads to smaller automata, which again suggests a connection between monitor size and monitor efficiency.

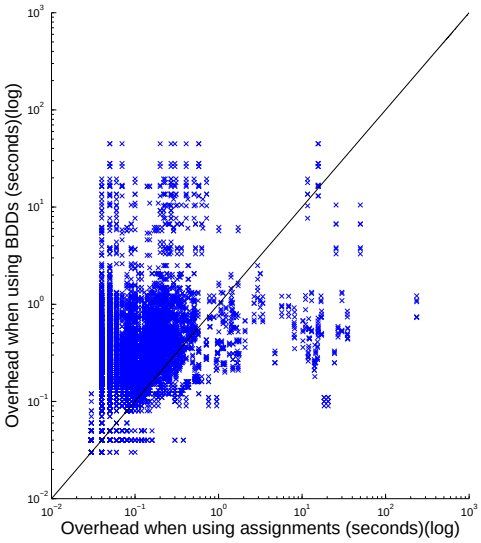

Fig. 6 Using assignments for alphabet representation leads to better performance than using BDDs. Points fall above the diagonal when assignment-based representation is better.

7.3 Alphabet minimization

Our data shows that partial– and full– alphabet minimization typically slow down the monitor (see Figure 7). We think that the reasons behind this are two-fold. On one hand, the performance of `gcc` as a decision engine to discover mutually exclusive conjunctions is not very good (in our experiments it was able to discover only 10%–15% of the possible mutually exclusive conjunctions). On the other hand, augmenting the formula increases the formula size, but SPOT does not take advantage of the extra information in the formula and typically generates bigger Büchi automata. If we manually augment the formula with *all* mutually exclusive conjunctions we do see smaller Büchi automata, so we believe this optimization warrants further investigation.

7.4 Monitor encoding

Finally, we compared the effect of the different monitor encodings (Fig. 8). Our conclusion is that no encoding dominates the others, but two (`front_nondet`

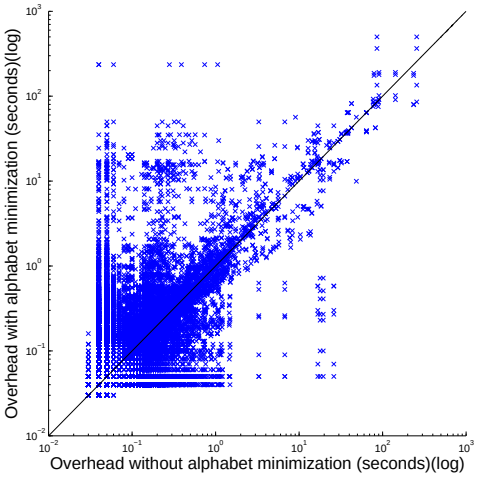

Fig. 7 Effect of alphabet minimization on monitor overhead. Points fall below the diagonal when alphabet minimization results in lower overhead. We do not see a significant advantage to using alphabet minimization, but this may be due to the particular tool chain that we used.

and `front_det_switch`) show the best performance relative to all others, while `back_det` has the worst performance. Comparing `front_nondet` and `front_det_switch` directly to each other (Fig. 9) indicates that `front_det_switch` delivers better performance for all but a few formulas.

7.5 Best non-table-based workflow

The final check of our conclusion is presented in Figure 10, where we plot the performance of the winning workflow against all other workflows. There are a few outliers, but overall the workflow gives better performance than all others.

Based on the comparison of individual optimizations we conclude that `front_det_switch` encoding with assignment–based state minimization and no alphabet minimization is the best overall workflow.

8 Results for table-based workflows

Soon after we completed the experiments described in Section 7, the compute cluster Ada was decommissioned, thus preventing us from evaluating the table-based encodings on the same hardware. In order to make an objective comparison between the different encodings, we re-ran all original experiments and new experiments involving the table-based encodings, on the Shared University Grid at Rice (SUG@R), Rice's Intel Xeon compute cluster.[9] Each of

[9] `http://rcsg.rice.edu/sugar/`

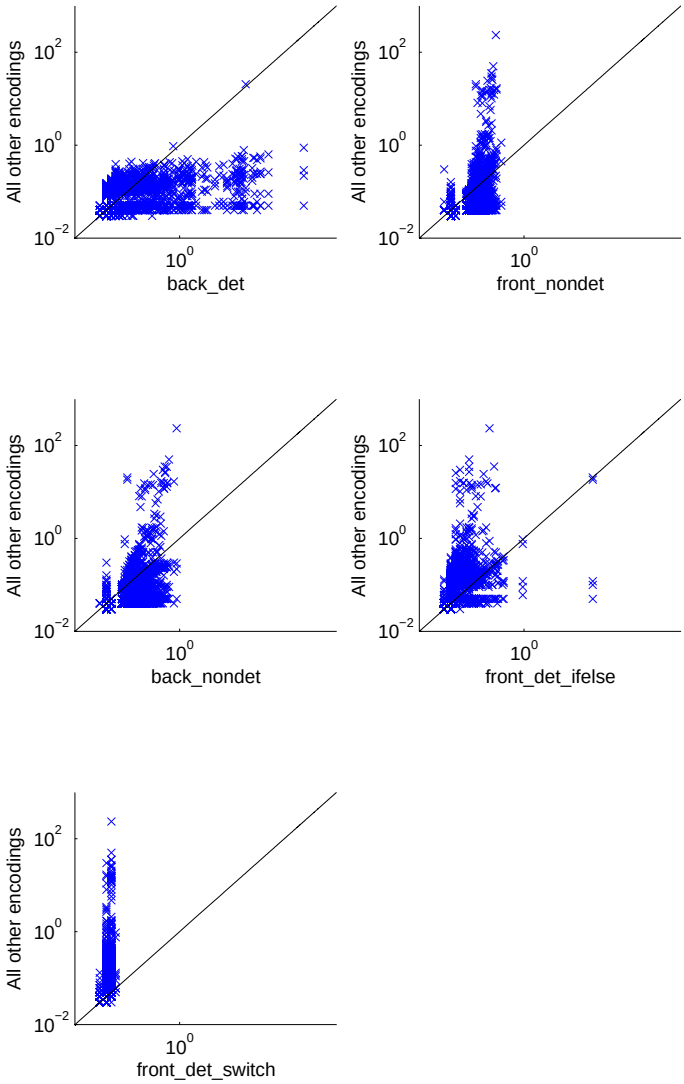

Fig. 8 Comparison of the monitor overhead when using different encodings. Each subplot shows the performance when using one of the encodings (x-axis) vs. all other encodings (y-axis). Points fall above the diagonal when the featured encoding results in lower overhead.

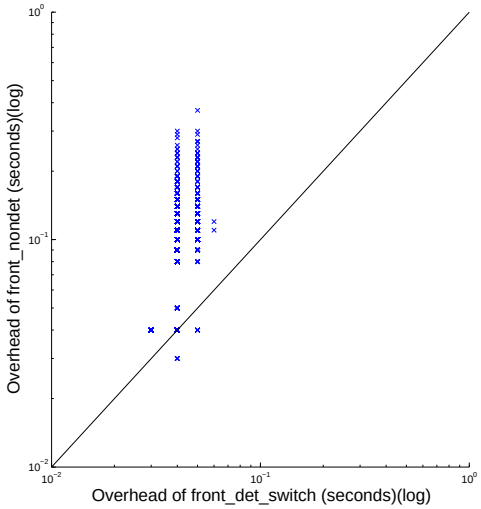

Fig. 9 Comparison of the monitor overhead when using the two best encodings (`front_det_switch` and `front_nondet`). Points fall above the diagonal when we see better performance using `front_det_switch`, which is the case for all but a few formulas.

SUG@R's 134 SunFire x4150 nodes has two quad-core Intel Xeon processors running at 2.83GHz and 16GB of RAM per processor. SUG@R is running Red Hat Enterprise 5 Linux, 2.6.18 kernel. We ran with exclusive access to a node so all 16GB of RAM were available for use. As before, we allowed 8 hours of computation time per workflow per formula for generating Büchi automata, automata-theoretic transformations, generating `C++` code, compilation, linking with the Adder model using the monitoring framework presented in [41], and executing the monitored model 10 times.

First we confirmed that the conclusions based on the initial experiments on Ada remain valid when applied to the experimental results obtained on SUG@R. For example, we compared the performance of the winning workflow identified in Section 7 against the performance of the 27 non-table workflows. The results are presented in Fig. 11. We observe that the `front_det_switch` encoding with assignment–based state minimization and no alphabet minimization dominates the other non-table-based workflows on SUG@R, thus validating our earlier conclusion.

Next we consider the performance of the two table-based workflows. Each was run on the same set of formulas as the other workflows. First we show the runtime overhead when using the file-based table encoding, compared to the overhead of all non-table-based encodings (Fig. 12). Although for some formulas the file-based table encoding shows significantly smaller overhead, for others it shows much larger overhead. Our interpretation of this data is that the cost of accessing the disk to read the file containing the automaton incurs an overhead that cannot be offset by the workflow's runtime performance.

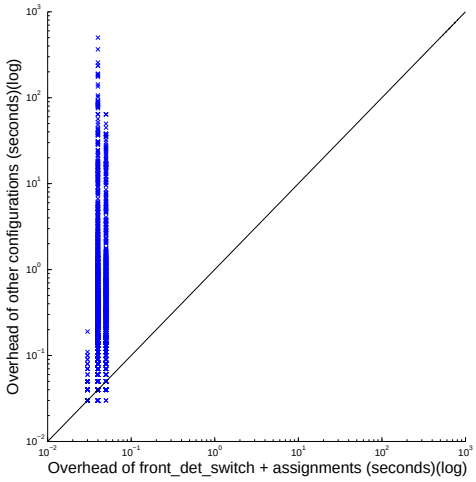

Fig. 10 Best overall performance for non-table-based workflows. Points fall above the diagonal when the `front_det_switch` encoding, with minimization, assignment-based alphabet representation, and without alphabet minimization results in lower monitor overhead.

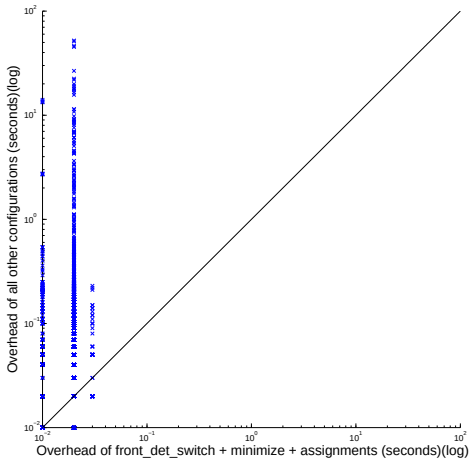

Fig. 11 Comparison of the monitor overhead when using the `front_det_switch` encoding with assignment–based state minimization and no alphabet minimization (x-axis) vs. all other encodings (y-axis). Points fall above the diagonal when the winning workflow identified on Ada also dominates the (non-table-based) workflows when executing on SUG@R.

We evaluate the performance of the memory-based table encoding in a similar manner (Fig. 13). We see that avoiding disk access improves the performance significantly over the file-based table encoding. This observation is confirmed by direct comparison between file-based and memory-based ta-

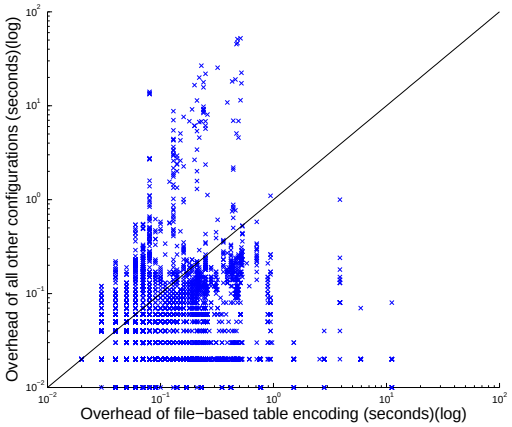

Fig. 12 Comparison of the monitor overhead when using the `front_det_file_table` encoding (x-axis) vs. all other encodings (y-axis). Points fall above the diagonal when the `front_det_file_table` encoding results in lower monitor overhead.

ble encoding (Fig. 14). For all formulas evaluated by the two workflows, the memory-based encoding is at least as fast (in most cases, significantly faster) than the file-based encoding.

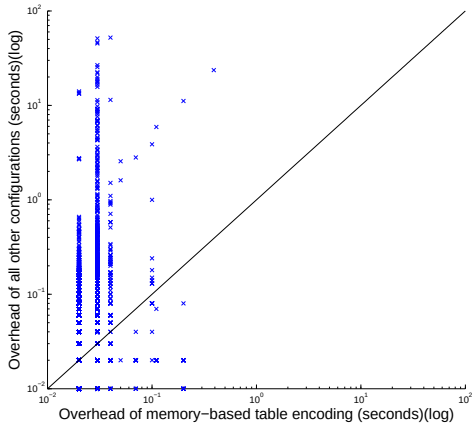

Fig. 13 Comparison of the monitor overhead when using the `front_det_memory_table` encoding (x-axis) vs. all other encodings (y-axis). Points fall above the diagonal when the `front_det_memory_table` encoding results in lower monitor overhead.

This data indicates that the memory-based table encoding is very competitive, but it is not clear whether its performance is better than the wining workflow identified in Section 7. Direct comparison of the runtime overhead is

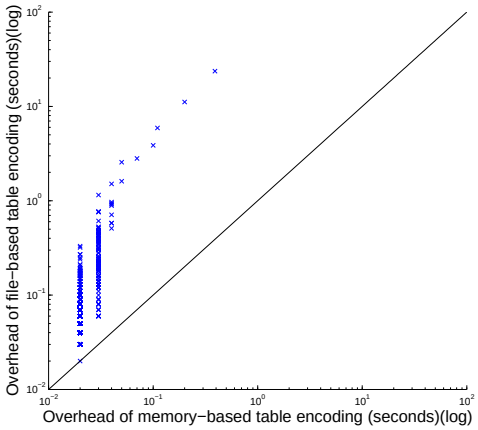

Fig. 14 Comparison of the monitor overhead when using the `front_det_memory_table` encoding (x-axis) vs. the `front_det_file_table` (y-axis). Points fall above the diagonal when the memory-based table encoding results in lower overhead.

presented in Fig. 15. Our conclusion is that for the majority of formulas the runtime overhead of the winning workflow identified earlier is smaller. Thus, the `front_det_switch` encoding with assignment–based state minimization and no alphabet minimization remains the best overall workflow that we have evaluated.

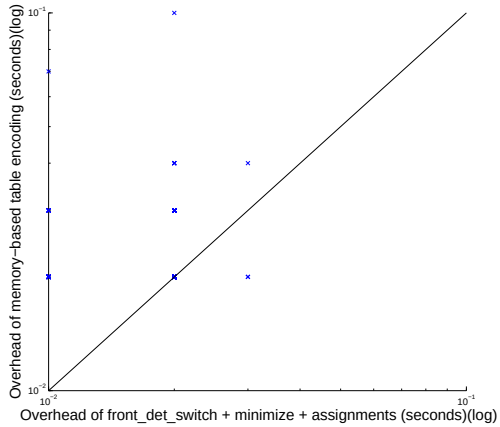

Fig. 15 Comparison of the monitor overhead when using `front_det_memory_table` encoding (x-axis) vs.the best encoding identified earlier (y-axis). The latter shows better runtime overhead, indicated by points falling above the diagonal. Although there are more than 900 data points in this figure, most of them are on top of each other.

9 Discussion and future work

In this paper we focus on minimization of monitor runtime. We identify the exploration space consisting of monitor encodings, alphabet encodings, transition representation, and other possible optimizations. We use off-the-shelf components (SPOT, `BRICS Automaton`, `gcc`) to perform some of the transformations, and a custom tool (CHIMP) to manage the different workflows. Together with the specification formalism proposed in [42], and the monitoring framework described in [41], this work provides a general ABV solution for temporal monitoring of SystemC models. Since the starting point is $\mathcal{A}_{NBW}(\psi)$, the techniques presented here are easy to integrate with a wide variety of specification languages. For example, it is easy to see that by applying [9] we can easily extend the scope of this work to efficient monitoring of PSL properties. We have identified a workflow that generates low-overhead monitors and we believe that it can serve as a good default setting.

Although the two table-based workflows have higher runtime overhead, they offer other important advantages. Both table-based workflows allow us to reduce the size of the monitor from hundreds of thousands of lines of code in some cases, to hundreds of lines of code. This avoids compilation problems and reduces the compilation time significantly. Another advantage of using the file-based table encoding is the flexibility to change the monitored properties without recompiling the MUV. The focus of this paper is on runtime overhead and exploring these issues is beyond its scope, but we believe that they are worthy of further consideration.

Practical use of our tool may involve monitoring tasks that are different than the synthetic load that we used for our tests. Recent developments in the area of self-tuning systems show that even highly optimized tools can be improved by orders of magnitude using search techniques over the workflow space (c.f., [27]). One possible extension of our work is to apply different optimizations to different types of formulas. For example, our data shows that when the minimized automaton ($\mathcal{A}^{bdd}_{DFW}(\psi)$ or $\mathcal{A}^{abr}_{DFW}(\psi)$) has more states than the unminimized automaton ($\mathcal{A}_{NFW}(\psi)$), generating a monitor using $\mathcal{A}_{NFW}(\psi)$ leads to smaller runtime overhead. This observation can be used as a heuristic, and further investigation may reveal that for different classes of formulas different workflows yield the best results. Thus, we have left the user full control over the tool workflow.

Acknowledgements We thank Alexandre Duret-Lutz for his code patch replacing the functionality of `spot::prune_scc`, which we used to upgrade from SPOT 0.4, used in the experiments executed on Ada, to SPOT 0.7.1 used in the experiments executed on SUG@R. We also thank Patrick Meredith and Dmitry Korchemny for suggesting that we consider table look-up encodings. Finally, we thank the anonymous reviewers for their comments and feedback.

References

1. Abarbanel, Y., Beer, I., Gluhovsky, L., Keidar, S., Wolfsthal, Y.: Focs: Automatic generation of simulation checkers from formal specifications. In: CAV'00: Proc. of the 12th International Conference on Computer Aided Verification, pp. 538–542 (2000)
2. Armoni, R., Korchemny, D., Tiemeyer, A., Vardi, M., Zbar, Y.: Deterministic dynamic monitors for linear-time assertions. In: Proc. Workshop on Formal Approaches to Testing and Runtime Verification, *Lecture Notes in Computer Science*, vol. 4262. Springer (2006)
3. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: FSTTCS'06: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, volume 4337 of LNCS, pp. 260–272. Springer (2006)
4. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. J. Log. Comput. **20**(3), 707–723 (2010)
5. Boulé, M., Zilic, Z.: Generating Hardware Assertion Checkers. Springer Publishing Company, Incorporated (2008)
6. Bryant, R.: Graph-based algorithms for Boolean-function manipulation. IEEE Trans. on Computers **C-35**(8) (1986)
7. Bryant, R.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24**(3), 293–318 (1992)
8. Bunker, A., Gopalakrishnan, G., McKee, S.A.: Formal Hardware Specification Languages for Protocol Compliance Verification. ACM Transactions on Design Autom. of Elec. Sys. **9**(1), 1–32 (2004)
9. Bustan, D., Fisman, D., Havlicek, J.: Automata construction for PSL. Tech. rep., The Weizmann Institute of Science (2005)
10. Chang, H., Cooke, L., Hunt, M., Martin, G., McNelly, A.J., Todd, L.: Surviving the SOC revolution: a guide to platform-based design. Kluwer Academic Publishers, Norwell, MA, USA (1999)
11. Chen, F., Jin, D., Meredith, P., Roşu, G.: Monitoring oriented programming - a project overview. In: Proceedings of the Fourth International Conference on Intelligent Computing and Information Systems (ICICIS'09), pp. 72–77. ACM (2009)
12. Cooper, K.D., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2004)
13. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. Formal Methods in System Design **1**, 275–288 (1992)
14. d'Amorim, M., Roşu, G.: Efficient monitoring of ω-languages. In: Proc. 17th International Conference on Computer Aided Verification, pp. 364–378 (2005)
15. Daniele, M., Giunchiglia, F., Vardi, M.Y.: Improved automata generation for linear temporal logic. In: CAV '99: Proc. 11th Int. Conf. on Computer Aided Verification, pp. 249–260. Springer-Verlag, London, UK (1999)
16. Duret-Lutz, A., Poitrenaud, D.: SPOT: An extensible model checking library using transition-based generalized Büchi automata. Modeling, Analysis, and Simulation of Computer Systems **0**, 76–83 (2004). DOI http://doi.ieeecomputersociety.org/10.1109/MASCOT.2004.1348184
17. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer, New York, Inc., Secaucus, NJ, USA (2006)
18. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Form. Methods Syst. Des. **24**(2), 101–127 (2004). DOI http://dx.doi.org/10.1023/B:FORM.0000017718.28096.48
19. Geilen, M.: On the construction of monitors for temporal logic properties. Electr. Notes Theor. Comput. Sci. **55**(2) (2001)
20. Geist, D., Biran, G., Arons, T., Slavkin, M., Nustov, Y., Farkas, M., Holtz, K., Long, A., King, D., Barret, S.: A methodology for the verification of a "system on chip". In: DAC '99, Proc. 36th Design Automation Conference, pp. 574–579. ACM, New York, NY (1999). DOI http://doi.acm.org/10.1145/309847.310001
21. Geldenhuys, J., Hansen, H.: Larger automata and less work for LTL model checking. In: In Model Checking Software, 13th Int. SPIN Workshop, volume 3925 of LNCS, pp. 53–70. Springer (2006)

22. Gerth, R., Peled, D., Vardi, M., Wolper, P.: Simple on-the-fly automatic verification of Linear Temporal Logic. In: P. Dembiski, M. Sredniawa (eds.) Protocol Specification, Testing, and Verification, pp. 3–18. Chapman & Hall (1995)
23. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Int. conf. on Automated Software Engineering, p. 412. IEEE, Washington, DC, USA (2001)
24. Grotker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers, Norwell, MA, USA (2002)
25. working group, I..: Standard for property specification language (PSL). IEC 62531:2007 (E) pp. 1–156 (2007). DOI 10.1109/IEEESTD.2007.4408637
26. Gupta, A.: Assertion-based verification turns the corner. IEEE Design and Test of Computers **19**, 131–132 (2002). DOI http://doi.ieeecomputersociety.org/10.1109/MDT.2002.10025
27. Hoos, H.H.: Computer-aided design of high-performance algorithms. Tech. rep., University of British Columbia (2008)
28. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (1979)
29. Jard, C., Jeron, T.: On-line model-checking for finite linear temporal logic specifications. In: Automatic Verification Methods for Finite State Systems, Proc. International Workshop, Grenoble, vol. 407, pp. 189–196. LNCS, Springer-Verlag, Grenoble (1989)
30. Jin, D., Meredith, P., Griffith, D., Roşu, G.: Garbage collection for monitoring parametric properties. In: Programming Language Design and Implementation (PLDI'11), pp. 415–424. ACM (2011). DOI doi:10.1145/1993316.1993547
31. Kupferman, O., Lampert, R.: On the construction of fine automata for safety properties. In: ATVA'06: Proc. of the International Symposium on Automated Technology for Verification and Analysis, pp. 110–124 (2006)
32. Kupferman, O., Vardi, M.: Model checking of safety properties. Formal methods in System Design **19**(3), 291–314 (2001)
33. Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Techniques for Technology Transfer (2011). To appear; http://dx.doi.org/10.1007/s10009-011-0198-6
34. Møller, A.: dk.brics.automaton (2004). Http://www.brics.dk/automaton/
35. Morin-Allory, K., Borrione, D.: Proven correct monitors from PSL specifications. In: DATE'06: Proc. Conf. on Design, automation and test in Europe, pp. 1246–1251. European Design and Automation Association (2006)
36. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, pp. 337–340 (2008)
37. Pierre, L., Ferro, L.: A tractable and fast method for monitoring SystemC TLM specifications. IEEE Transactions on Computers **57**, 1346–1356 (2008). DOI http://doi.ieeecomputersociety.org/10.1109/TC.2008.74
38. Rönkkö, M.: LBT: LTL to Büchi conversion. Available online (1999). URL `http://www.tcs.hut.fi/Software/maria/tools/lbt/`. Accessed March 29, 2011
39. Rozier, K.Y., Vardi, M.Y.: LTL satisfiability checking. In: Proc. 14th Int. SPIN conference on Model checking software, pp. 149–167. Springer, Berlin, Heidelberg (2007)
40. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. Electron. Notes Theor. Comput. Sci. **144**(4), 109–124 (2006). DOI http://dx.doi.org/10.1016/j.entcs.2006.02.007
41. Tabakov, D., Vardi, M.: Monitoring temporal SystemC properties. In: Proc. 8th Int'l Conf. on Formal Methods and Models for Codesign, pp. 123–132. IEEE (2010)
42. Tabakov, D., Vardi, M., Kamhi, G., Singerman, E.: A temporal language for SystemC. In: FMCAD '08: Proc. Int. Conf. on Formal Methods in Computer-Aided Design, pp. 1–9. IEEE Press (2008). URL `http://portal.acm.org/citation.cfm?id=1517446`
43. Tabakov, D., Vardi, M.Y.: Experimental evaluation of classical automata constructions. In: LPAR'05, 12th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, pp. 396–411 (2005)
44. Vardi, M., Wolper, P.: Reasoning about infinite computations. Information and Computation **115**(1), 1–37 (1994)
45. Vijayaraghavan, S., Ramanathan, M.: A Practical Guide for SystemVerilog Assertions. Springer, New York, NY, USA (2005)