Using Different Representations of Synchronous Systems in SAL

Manuel Gesell, Felipe Bichued, and Klaus Schneider
TU Kaiserslautern
gesell@cs.uni-kl.de
bichued@rhrk.uni-kl.de
schneider@cs.uni-kl.de

Abstract

In general, synchronous systems can be represented as a set of so-called synchronous guarded actions (SGAs) that consist of a trigger condition and an atomic action. Whenever the trigger condition holds, i.e., the guarded action is enabled, then the action is immediately executed. While the synchronous semantics demands that *all enabled* actions have to be executed concurrently within the same variable environment, it is possible for certain sets of guarded actions to deviate from the synchronous execution scheme without changing the behavior. This is important to make use of tools like SRI's Symbolic Analysis Laboratory (SAL) that work with invariants and guarded actions, but only a subset of the enabled actions are chosen for execution. If the particular choice of the enabled guarded actions for execution is not determined, we may consider different choices that might influence the resource requirements needed for formal verification. In this paper, we therefore investigate how three possible representations influence the runtime and memory requirements of automatic verification runs of SRI's SAL.

1. Introduction

The synchronous model of computation [Hal93, BCE+03] has proved to be very convenient for the development of reactive embedded systems. In particular, this is the case for systems consisting of application-specific hardware *and* software since compilers can generate hardware as well as software from the same synchronous programs. In general, a synchronous system performs its execution in discrete reaction steps that are considered as clock ticks of a global clock. In each reaction, *all* input values are read, and depending on these values, *all* output values are determined in addition to the change of the internal state of the system.

Synchronous languages like Esterel [BG92] and Quartz [Sch09] offer the explicit notion of reaction steps and many convenient statements for the design of reactive systems. The explicit notion of (logical) time also requires different kinds of assignments like immediate assignments that assign values in zero time and delayed assignments that assign values with a delay of one time unit.

For safety-critical applications, an important advantage of synchronous languages is the availability of a precisely defined formal semantics. This allows one to translate programs to state transition systems and to use formal verification techniques. Model checking procedures [GV08] are already established and are even integrated within the compilers, e.g., to check for instantaneous loop bodies, to guarantee the absence of write conflicts and runtime errors, to solve causality problems, and many other issues that might appear during compilation. However, it is well-known that the particular system representation is a crucial point for a formal verification task. For example, even a bad variable ordering in a BDD-based model checker may lead to an exponential blow-up in run time or memory usage while a better ordering could work efficiently.

In this paper, we therefore explore the possibilities of representing a synchronous system in SRI's Symbolic Analysis Laboratory (SAL), and evaluate their effect on the performance of SAL's model checker. To this end, we start in all cases with synchronous guarded actions as a general system representation. Since SAL only supports interleaved guarded actions, we already presented in [GS13] a possible translation of *synchronous guarded actions (*SGAs*)* to SAL's *interleaved guarded actions* (*IGAs*) to demonstrate that SAL can also be used to verify synchronous systems. This paper aims at comparing that system representation against alternatives with respect to the performance of the later model checking. Therefore, the approach presented in [GS13] (GC) will be compared with two others: One based on SAL's synchronous composition of modules (SC) and another one based on a translation to equation systems (ES). Therefore, we implemented the tool *aif2sal* that is capable of generating these representations from an *SGAs* description of a Quartz program (see Figure 1).

The three transformations GC, SC, and ES are based on different paradigms that lead to different computations of a macro step in SAL. The ES representation describes the behavior of all *SGAs* in a single transition step by describing them as invariants. Therefore, no guarded action is required and an equation system must be solved in each reaction step. The SC transformation models the behavior of each variable by a single module containing all guarded actions writing this variable. The synchronous model of computation assures that only a single guarded action defines the value of a variable in a reaction step. Hence, in each reaction step all modules execute a single guarded action synchronously to define the behavior. This obliges SAL to resolve the data-dependencies between the modules. The GC approach describes the behavior in a single module by *IGAs*. This requires the explicit modeling of the data-dependencies as described in [GS13]. Unlike the other approaches, the behavior of a single reaction step of the original system requires several transition steps in GC.

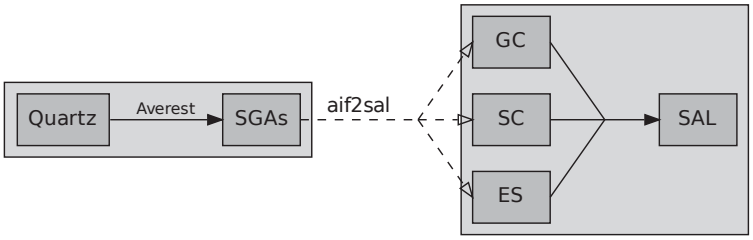

Figure 1: Averest/SAL linkage

The paper is organized as follows: the next section describes some preliminaries like the synchronous model of computation, the Averest tool-kit and SRI's SAL tool. Then, Section 3 presents related work we found in the literature. The main part of the paper is presented in Section 4, where

the above three representations are described. In Section 5, we present experimental results to compare the considered transformations and argue about the most advanced representation. Finally, we conclude the paper and discuss future work.

2. Preliminaries

In this section, we describe the synchronous model of computation, the Averest system and SAL.

2.1. Synchronous Models of Computation

The execution of synchronous languages [Hal93, BCE+03] is divided into a discrete sequence of reaction steps that are also called macro steps. Within each macro step, the system reads all inputs and instantaneously generates all outputs together with the next internal state depending on the current internal state and the read inputs. To compute the outputs and the next internal state, macro steps are divided into finitely many micro steps. Micro steps are atomic actions of the programs like assignments to variables. Since all micro steps of a macro step are executed in the same variable environment given by their reaction/macro step, all variables have unique values in each macro step. It may be the case that the trigger condition of an action depends on a variable that is modified by the action itself. Such cyclic dependencies are considered in the causality analysis for synchronous programs that checks whether for all inputs, the outputs can be determined in an order that respects the data dependencies.

2.2. Averest

Averest[1] is a framework for specification, implementation and verification of reactive systems and is developed by our group. The programs are written using the synchronous programming language Quartz and are translated to the Averest Intermediate Format (AIF), which is a representation of the program's behavior in terms of *SGAs*. An *SGA* $\langle \gamma \Rightarrow \alpha \rangle$ contains a boolean guard γ and an atomic assignment α that is executed whenever γ holds. Assignments can be either immediate $\langle \gamma \Rightarrow \texttt{x=}\tau \rangle$ or delayed $\langle \gamma \Rightarrow \textbf{next}\texttt{(x)=}\tau \rangle$. Both evaluate the right-hand side τ in the current step. While the immediate assignment transfers the obtained value already in the current step to the left-hand side `x`, the delayed assignments transfer the value only in the next macro step to `x`.

A simple module called ABRO can be seen on the left-hand side of Figure 2. This module has three inputs (indicated by **?**) `a`, `b` and `r`, and one output `o` (indicated by **!**). The program waits for the input events `a` and `b` and immediately emits output `o` as soon as the last one of `a` and `b` occurred. This behavior can be restarted with the reset input `r`.

The program contains two specifications `s1` and `s2`, where the first asserts that either `a` or *b* holds whenever *o* is emitted. Property `s2` asserts that `o` does not hold in successive points of time.

The ABRO module is compiled to the *SGAs* shown on the right-hand side of Figure 2. As can be seen, the guarded actions are separated into control flow and data flow. Control flow actions are assignments to control flow locations and data flow actions are assignments to local and output variables. Apart from the control flow locations `wa`, `wb` and `wr`, a new location `w0` (often called *boot-location*) is added by the compiler to serve as initial state.

[1] http://www.averest.org

```
module ABRO(event ?a,?b,?r,!o) {
   loop
      abort {
         wa: await(a);
         ||
         wb: await(b);
         emit(o);
         wr: await(r);
      } when(r);
} satisfies {
   s1 : assert A G (o ⇒ a ∨ b);
   s2 : assert A G (o ⇒ X ¬o);
}
```

```
system ABRO :
interface :
    a, b, r : input event bool
          o : output event bool
locals :
    w0, wa, wb, wr : label bool
synchronous guarded actions :
    control flow:
        True ⇒next(w0)=True
        ¬w0 ⇒next(wa)=True
        ¬w0 ⇒next(wb)=True
        ¬r∧wa∧¬a∨r∧(wr∨wa∨wb)⇒next(wa)=True
        ¬r∧wb∧¬b∨r∧(wr∨wa∨wb)⇒next(wb)=True
        ¬r∧(wr∨a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
          ⇒next(wr)=True
    data flow:
        ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)⇒o=True
specifications:
    s1: A G o → a ∨ b
    s2: A G o → X ¬o
```

Figure 2: ABRO Example

2.3. SAL

SRI's Symbolic Analysis Laboratory[2] is a framework intended for performing abstraction, program analysis and model checking, and it provides an intermediate language which will be the target of our translation process. A typical SAL system is represented by a context containing a set of modules and assertions. Each module declares a distinct set of inputs, outputs, local and global variables, as well as definitions (invariants) and transitions. Variables can be either current (X) or next variables (X'), where assignments to current variables take place in the current state, and to next variables in the following state. SAL allows to compose modules either synchronously ($||$) or asynchronously ($[]$). In synchronously composed modules, a transition from each module is executed simultaneously. With asynchronous composition however, an enabled transition from exactly one module is executed non-deterministically. Transitions can be written as an equation or as guarded commands. The equational format defines the trajectory of single variables, while the guarded commands define single transitions in the system. A guarded command of SAL contains, contrary to *SGAs*, a set of assignments and is enabled when its guard evaluates to true. Furthermore, SAL non-deterministically picks one of the enabled guarded commands and updates the next-state variables accordingly. The structure and behavior is equivalent to *IGAs* described in [GS13]. A system without enabled guarded commands leads to a deadlock. A large set of tools such as symbolic/bounded model checkers, simulators and others, can then be used for analysis and verification.

3. Related Work

The idea of transforming *SGAs* to other models of computation was already done before: an automatic translation to SystemC by generating a dynamic schedule for modules in order to preserve the semantics was presented in [BGS10]; and in [BBS11] *SGAs* are translated to asynchronous DPNs.

[2] http://sal.csl.sri.com/

There exists also a similar approach [SB08, BS08] that refines the translation of *SGAs* to transition systems at the level of micro steps. The intention of [SB08, BS08] was to use these transition systems at the micro step level to perform causality analysis by means of theorem proving and bounded model checking. We follow similar ideas in the GC transformation, but work at a different level of abstraction, and furthermore, our approach allows us to improve the system representation by handling e.g. the default reaction in a different way.

The differences between *SGAs* and Hoare's parallel commands [Hoa78] are that *SGAs* do not have a disjoint set of variables and they communicate over shared variables (broadcast).

In contrast to Dijkstra's guarded commands [Dij75], our *IGAs* have only a single *repetitive construct* consisting of the entire set of *IGAs*. Hence, our GC translation targets a subset of Dijkstra's guarded commands. This is justified since we do not wish to use the guarded actions as primary input language, and use them rather as intermediate representation.

The Z2SAL project [DNS06, DNS11] connects Z to SAL by defining a transformation to SAL's input language. Additionally, we compare different approaches of representing Quartz in SAL. One of our transformations uses the built-in synchronous-composition operator of SAL similar to [PSSD00] where dynamic constructs were used to embed the behavior of multi-threaded Java programs in SAL.

4. Different Representations of Synchronous Systems

In this section, we present three different approaches of describing synchronous systems in SAL's input language. To ease the translation process, we have developed a tool called *aif2sal*, which is capable of converting AIF to the three different representations in SAL.

4.1. Guarded Commands GC

Guarded commands in SAL are interpreted as interleaved guarded actions (*IGAs*), meaning that in each transition step an enabled guard action is non-deterministically chosen to define the step's behavior. This is very different from the way that *SGAs* work and thus many problems have to be solved when representing a synchronous program as *IGAs*. These problems are described in detail in [GS13] and will be summarized in the following.

By executing the guarded actions in an interleaved fashion, a data dependency problem appears because of the non-deterministic choice of a single action. The *IGAs* now can execute in arbitrary orders, including those where a value is not yet present for a specific variable. Furthermore, the temporal behavior of a synchronous program is violated due to the increased number of steps it takes to complete a macro step.

$$\begin{array}{lcl|lcl} \gamma_1 & \Rightarrow & \mathtt{x} = \tau_1 & \delta_1 & \Rightarrow & \mathbf{next}(\mathtt{x}) = \upsilon_1 \\ & \vdots & & & \vdots & \\ \gamma_n & \Rightarrow & \mathtt{x} = \tau_n & \delta_m & \Rightarrow & \mathbf{next}(\mathtt{x}) = \upsilon_m \end{array}$$

Figure 3: Guarded Actions for variable `x`

The key to solve these problems is to modify the guards such that the data dependencies are explicitly stated. Hence, the solution of [GS13] represents each *SGA* containing an immediate assignment by separate *IGAs*, and all *SGAs* containing a delayed assignment are composed to a single

IGA called *conclusion*. Additionally, we need to introduce for all variables written by immediate assignments a new variable, the valid flag $\mathtt{x}_v$, that determines the validity of the value contained in the variable and is used to deactivate all *IGAs* writing to the corresponding variable x once a value is determined in the current macro step. Hence, the synchronous guarded actions for the variable x in Figure 3 are converted into the following *IGAs*:

$$
\begin{array}{lcl}
\neg \mathtt{x_v} \wedge \left(\bigwedge_{v \in \mathsf{read}(\gamma_1 \Rightarrow \mathtt{x} = \tau_1)} v_\mathrm{v} \right) \wedge \gamma_1 & \Rightarrow & \left[\begin{array}{ll} \mathtt{x} & = \tau_1 \\ \mathtt{x_v} & = \mathsf{true} \end{array} \right] \\
\vdots & & \\
\neg \mathtt{x_v} \wedge \left(\bigwedge_{v \in \mathsf{read}(\gamma_n \Rightarrow \mathtt{x} = \tau_n)} v_\mathrm{v} \right) \wedge \gamma_n & \Rightarrow & \left[\begin{array}{ll} \mathtt{x} & = \tau_n \\ \mathtt{x_v} & = \mathsf{true} \end{array} \right] \\
\neg \mathtt{x_v} \wedge \left(\bigwedge_{i=1 \ldots n} \neg \gamma_i \right) & \Rightarrow & \left[\begin{array}{ll} \mathtt{x_v} & = \mathsf{true} \end{array} \right]
\end{array}
\quad \Bigg| \quad
\bigwedge_{v \in \mathcal{V}} v_\mathrm{v} \Rightarrow
\left[\begin{array}{l}
\vdots \\
x = \begin{cases} v_1, & \text{if } \delta_1 \\ \vdots & \\ v_m, & \text{if } \delta_m \\ \text{defaultAct (x)}, & \text{else} \end{cases} \\
x_\mathrm{v} = \bigvee_{i=1 \ldots m} \delta_i \\
\vdots
\end{array} \right]
$$

Whenever we determined a valid value for x by enabling $\mathtt{x}_v$, all *IGAs* writing x are automatically deactivated by the term $\neg \mathtt{x_v}$. The following term ensures that all values required to evaluate the *IGA* are valid. The original behavior is still encoded in γ_i and the assignment $\mathtt{x} = \tau_i$. Additionally, the assignment $\mathtt{x_v} = \mathsf{true}$ indicates the validity of x for the current macro step. In case none of the guards hold, no assignment for that variable takes place in the current macro step and thus it must contain the default value (e.g. the value of the previous step) – which also means that $\mathtt{x}_v$ must be enabled. Therefore, the default value of the variable should be already contained in the variable x. This is ensured by the *conclusion* (right-hand side) that executes for all variables the delayed assignments. There, the default value must be assigned to all variables not written by a delayed assignment. This is possible, because the value of a variable x is not used unless $\mathtt{x_v}$ holds. Additionally, the *conclusion* initiates the execution of the next macro step by resetting the valid flags of all variables (not written by delayed assignments).

The SAL GC-representation of our running example has the structure shown in Figure 4. One can see that only a single valid flag (for the variable o) is required, because all other variables are written by delayed assignments. Additionally, the specifications were adapted to cover the changed temporal behavior. All newly introduced immediate states have in common that not all variables have a valid value and so the adapted specification only requires that the original specification is satisfied in states where all variables contain valid values.

4.2. Synchronous Composition SC

Another idea is to exploit SAL's synchronous composition primitive and divide the program into a set of synchronous modules. To that end, each variable, with the exception of inputs, will be represented as an independent module, and these modules will be then composed synchronously to provide the overall system behavior. It is worth noting that the semantics of the synchronous composition closely matches that of the synchronous model of computation. Since every variable has a unique value in each reaction step determined by a single *SGA* every module will execute exactly one transition.

In contrast to the GC transformation, the SC transformation is just a syntactic rewrite of the original program, in the sense that no guarded action will be modified. In this approach, data

```
ABROGC: MODULE =
BEGIN
  INPUT a,b,r : BOOLEAN
  OUTPUT o_v, o : BOOLEAN
  LOCAL w0,wa,wb,wr,o_v : BOOLEAN
  INITIALIZATION [w0 = wa = wb = wr = o = o_v = FALSE]
  TRANSITION [
    [] ¬o_v ∧ ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa) ⟶
        o_v' = TRUE ;
        o' =  TRUE ;
    [] ¬o_v∧(r∨¬(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)) ⟶
        o_v' = TRUE ;
    [] o_v ⟶
        wr' =  ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
        wb' =  ¬r∧wb∧¬b∨r∨¬w0;
        wa' =  ¬r∧wa∧¬a∨r∨¬w0;
        w0' =  TRUE ;
        o' =  FALSE ;
        o_v' = FALSE ;
    ]
END;
s1 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ a ∨ b)];
s2 : THEOREM ABROGC ⊢ AG[¬o_v U (o ⇒ X¬o)];
```

Figure 4: GC Representation

dependencies are resolved internally by SAL. The synchronous composition combines all definitions, initializations and transitions of the composed modules, taking care that the combination is still casually correct. In case inconsistencies in the conjunction of the transitions are found, proof obligations are generated, but this problem does not apply here because the Averest compiler rules out causally incorrect programs.

The translation to SC consists of separating *SGAs* by their written variable into individual modules as depicted in Figure 5a (for variable *o*). Each module will have every other variable that is read by the *SGAs* as input and a single output being the writable variable itself. All *SGAs* for each variable are then collected, and used to properly initialize the module and to describe its transitions as guarded commands.

```
oMod : MODULE =
BEGIN
  INPUT  a, b, r, wa, wb : BOOLEAN
  OUTPUT o : BOOLEAN
  INITIALIZATION
    [o=¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)]
  TRANSITION
    [  ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa)
          ⟶ o' = TRUE;
    [] ELSE ⟶ o' = FALSE; ]
END
```

(a) SC: Single module

```
 w0Mod : MODULE = ...
 waMod : MODULE = ...
 wbMod : MODULE = ...
 wrMod : MODULE = ...
  oMod : MODULE = ...
ABROSC : MODULE = w0Mod
                  || waMod
                  || wbMod
                  || wrMod
                  ||  oMod;
s1 : THEOREM ABROSC ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABROSC ⊢ AG(o ⇒ X¬o);
```

(b) SC: Composition

Figure 5: Single Module and Synchronous Composition

In Figure 5b, we see how such a synchronous composition might look like. We simply compose all writable variables (`w0`, `wa`, `wb`, `wr` and `o`) into a single module. It is important to note that a composed module will be deadlocked whenever at least one of the modules is deadlocked, hence we

introduce an **ELSE** guard to guarantee that there is always a transition to be taken. Moreover, the **ELSE** guard will assign the default value to the variable, which is effectively the default reaction.

4.3. Equation System ES

This transformation converts the *SGAs* into equations (one per variable). It is usually not trivial to generate such an equation system, and a corresponding transformation is already implemented in the Averest system. The translation of *SGAs* to equations will generate exactly one equation for each output, local and location variable. Furthermore, an additional carrier variable must be added for each variable to which an immediate and delayed assignment is made. The carriers will simply hold the value until the next point of time.

The execution of an equation system under the synchronous model of computation is then as follows: in every macro step new input variables are read and all of the equations are evaluated with regards to the newly read values. The resulting right-hand side of each equation is then assigned to its respective variable.

This can be easily done in SAL by using definitions instead of guarded commands. Definitions in SAL are of the form $\langle X = \texttt{EXPR} \rangle$ for the current state or $\langle X' = \texttt{EXPR} \rangle$ for the next state. In contrast to guarded commands, which are picked individually, all definitions are evaluated in every state and the resulting value for the expression EXPR is assigned to the variable. Once we have the

```
ABRO_ES : MODULE =
BEGIN
  INPUT a, b, r : BOOLEAN
  OUTPUT o : BOOLEAN
  LOCAL wa, wb, wr, w0 : BOOLEAN
  INITIALIZATION [w0 = wa = wb = wr = FALSE]
  DEFINITION o = ¬r∧(a∧wa∧b∧wb∨¬wa∧b∧wb∨¬wb∧a∧wa);
  TRANSITION
    w0' = TRUE;
    wa' = ¬r∧wa∧¬a∨r∨¬w0;
    wb' = ¬r∧wb∧¬b∨r∨¬w0;
    wr' = ¬r∧(wr∨(a∧wa∧b∧wb)∨(b∧wb∧¬wa)∨(a∧wa∧¬wb));
END;
s1 : THEOREM ABRO_ES ⊢ AG(o ⇒ a ∨ b);
s2 : THEOREM ABRO_ES ⊢ AG(o ⇒ X¬o);
```

Figure 6: ES: Single module

SGAs given as equations, the translation to SAL is straightforward. We will have a single module containing the original inputs and outputs and all other variables as local variables, as seen on Figure 6. SAL supports in the **DEFINITION** section only assignments to variables of the current state, hence a distinction between **INITIALIZATION/TRANSITION** and **DEFINITION** is necessary. All equations defining a next-state variable must be initialized in the **INITIALIZATION** section and described in the **TRANSITION** section. Hence, the immediate assignments will be represented as an invariant in the **DEFINITION** section and will be evaluated in every state, including the initial one. The **INITIALIZATION** section will be evaluated in the initial state and contains the initialization of variables written by delayed assignments like the location variables and the

carriers[3] to their default values. The **TRANSITION** section contains equations for evaluating the next-state.

Note that problems like the default reaction were already handled during the translation to an equation system by simply adding an extra branch to every equation, assigning the variable's default value whenever none of the previous conditions hold.

5. Experimental Results

The presented transformations were used to verify and benchmark the experiments using SAL's symbolic model checker (sal-smc). All experiments[4] were performed on a Intel® Core™ i5-3470 CPU @ 3.20GHz using Ubuntu 13.04.

In the following, we briefly describe each example, in terms of what they do, number of variables in the original Quartz program, number of *SGAs* after compilation, and the number and kind of properties that were verified for each. This gives an idea on the complexity of each example:

P	#SGA	#GC	#SC	#ES	GC	SC	ES
ABRO	7	4	6(12)	5	0.11	0.06	0.05
ABROM[M=10]	23	2	14(36)	13	0.74	1.13	0.50
ABROM[M=13]	29	2	17(45)	16	4.27	7.92	3.27
AuntAgatha	2	4	3(4)	3	0.12	0.07	0.09
VendingMachine	23	23	12(32)	11	1.14	0.15	0.07
LightControl	36	25	12(47)	11	1.79	0.44	0.40
MinePumpController	42	41	22(61)	21	7.60	0.22	0.09
RSFlipFlop	7	2	5(11)	8	53.51	1.18	1.18
MemoryController	41	28	17(87)	31	407.95	42.93	3.42
IslandTrafficControl	83	62	35(109)	36	504.64	62.40	1.94

Figure 7: Size of the Representations and Execution Times (in sec) of SAL

ABROM is a larger version of the ABRO example which waits for M events in parallel instead of just two. It contains **22** *SGAs* for $M = 10$ or **28** for $M = 13$, **2** inputs, **1** output and **3** safety properties.

AuntAgatha is an implementation of an old puzzle where the reader has to find who killed Aunt Agatha in Dreadsburry Mansion based on simple boolean statements. The problem is represented by **2** *SGAs*, **21** inputs, **1** output and **3** boolean properties.

VendingMachine is a vending machine controller that dispenses gum in reaction to the insertion of nickels and dimes, which is described by **2** *SGAs*, **2** inputs, **2** outputs, and **3** safety properties.

LightControl models the light control system of a room with regards to to its occupancy. Its functions include switching the light on/off, dimmer control and notification of alarms. The implementation contains **36** *SGAs*, **22** inputs, **12** outputs, and **10** safety specifications.

[3] Carriers are present only when the program utilizes immediate and delayed assignments for a single variable, which is not the case for ABRO.

[4] All examples a publicly available under `http://www.Averest.org/examples`.

MinePumpController starts or stops the pump of a mine according to alerts issued by the carbon dioxide and methane monitors, as well as the water level. It contains **40** *SGAs*, **27** inputs, **30** outputs, and **7** safety specifications.

RSFlipFlop describes a RS-Flipflop with NOR-gates of equal delay, modeled as a single macro step. It contains **7** *SGAs*, **2** inputs, **2** outputs, and **8** specifications (three safety and five co-Büchi).

MemoryController models a memory controller providing mutual exclusion by maintaining region locks for addresses. The implementation contains **41** *SGAs*, **5** inputs, **12** outputs, and **8** safety specifications.

IslandTrafficControl: An island is connected via a tunnel with the mainland. Inside the tunnel is a single lane so that cars can either travel from the mainland to the island or vice versa, which is signaled by traffic lights on both ends of the tunnel. It is represented by **75** *SGAs*, **15** inputs, **32** outputs, and **13** specifications (eleven safety and two Büchi) modeled in **5** modules.

The table in Figure 7 roughly measures the size of the original program regarding the number of *SGAs* (#SGA), as well as the size of each representation in terms of the number of guarded commands (#GC) for GC, number of modules and guarded commands (#SC[5]) for SC, and the number of equations (#ES) for ES. Interestingly, ABROM contains only 2 guarded commands in GC, while having about 20 *SGAs* in the original Quartz program. This happens because ABROM only features delayed assignments, which according to the transformation described in 4.1, are combined into a single guarded command called *conclusion*. This also explains the particularly good performance for GC on verifying it (Figure 7). The number of equations used in ES corresponds with the number of modules used for SC. For each variable an equation is contained in ES and SC contains besides the module for the synchronous composition for each variable a module. In case a carrier variable has to be introduced for ES, they will differ.

P	GC		SC		ES	
	#V	#N	#V	#N	#V	#N
ABRO	34	786	30	616	14	183
ABROM[M=10]	80	2289	98	3855	52	1221
ABROM[M=13]	98	3339	122	3381	64	2065
AuntAgatha	100	1434	40	130	48	180
VendingMachine	130	10855	138	4496	22	384
LightControl	98	7108	114	6908	44	837
MinePumpController	126	98949	136	8500	36	636
RSFlipFlop	36	1883	38	929	24	1010
MemoryController	158	931422	188	92815	98	48134
IslandTrafficControl	144	773001	212	58217	58	30401

Figure 8: BDD size in terms of the number of variables (#V) and number of nodes (#N)

As for the actual performance of each representation, Figure 7 shows that ES is generally faster than GC or SC. In the worst case (IslandTrafficControl), it was more than 250 times faster than an equivalent program in the GC representation and roughly 32 times faster than SC. Not surprisingly, the complexity of the properties can also increase the verification time in certain cases, as with the

[5]The number inside parenthesis is the sum of the number of guarded commands in the context.