Enabling Dynamic Assertion-based Verification of Embedded Software through Model-driven Design

Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, Graziano Pravadelli
University of Verona - Department of Computer Science
Email: {name.surname}@univr.it

***Abstract*—Assertion-based verification (ABV) is more and more used for verification of embedded systems concerning both HW and SW parts. However, ABV methodologies and tools do not apply to HW and SW components in the same way: for HW components, both static ABV and dynamic ABV are widely used; on the contrary, SW components are traditionally verified by means of static ABV, because dynamic approaches are based on simulation assumptions which could not be true during execution of general embedded SW and which cannot be controlled by the assertion language. This paper proposes to exploit model-driven design for guaranteeing such simulation assumptions. Then, it describes an ABV framework for embedded SW, that automatically synthesizes assertion checkers to verify the embedded SW accordingly to the simulation assumptions.**

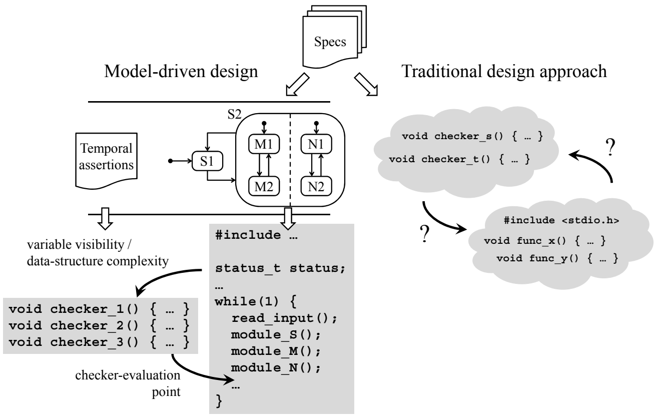
I. Introduction

In the recent years, *assertion-based verification* (ABV) [1] has assumed a significant role in verification of embedded systems, concerning both HW and SW parts. ABV relies on the definition of temporal assertions to verify the functional correctness of the design with respect to the expected behavior. Approaches based on ABV can be classified in two main categories: *static* (i.e, *formal*) and *dynamic* (i.e, *simulation-based*).

In static ABV, assertions, representing design specifications, are written as temporal logic formulas (e.g., LTL, CTL) and they are exhaustively checked against a formal model of the design by exploiting, for example, a model checker. Such an exhaustive reasoning provides the verification engineers with high confidence in system reliability. However, the well-known state-space explosion problem limits the applicability of static ABV to small/medium-size, high-budget and safety-critical projects [2].

On the contrary, thanks to the scalability provided by simulation-based techniques, dynamic ABV approaches are preferred for verifying large designs, which have both reliability requirements and stringent development-cost/time-to-market constraints. In dynamic ABV, assertions are defined by using a formal language, e.g., Property Specification Language (PSL) [3]. Then, they are compiled into *assertion checkers*, or simply *checkers*, i.e., modules that capture the behavior of the corresponding assertions and monitor, during simulation, if they hold with respect to the design [4]. In the HW domain, several verification approaches have been proposed by exploiting assertion checkers [1], [5], and dynamic ABV is affirming as a leading strategy in industry to guarantee fast and high-quality verification of HW components [6], [7]. On the contrary, companies, which develop embedded software (ESW), still incur practical problems in adopting dynamic ABV in their design flows [8].

At first, the application of ABV for ESW is hard to achieve due to the high skills required to define effective assertions.



```
#include …

status_t status;
…
while(1) {
  read_input();
  module_S();
  module_M();
  module_N();
  …
}
```

```
void checker_1() { … }
void checker_2() { … }
void checker_3() { … }
```

```
void checker_s() { … }
void checker_t() { … }
```

```
#include <stdio.h>
void func_x() { … }
  void func_y() { … }
```

Fig. 1: Integration of dynamic ABV in a MDD approach vs. a traditional (manual) approach.

Project managers encourage SW architects and developers to write assertions, but they have problems with learning PSL, translating informal specifications into assertions, and mastering assertion coding. To overcome such difficulties, predefined libraries of assertion checkers have been proposed in the HW domain, like, for example, the Open Verification Library (OVL) [9], but, to the best of our knowledge, nothing similar is available for ESW. Secondly, existing checker generators are limited to the hardware domain and cannot be easily integrated in an ESW design flow [4], [10], [11]. This is mainly due to the characteristics of ESW, which generally differs from HW descriptions in several aspects. For example, HW descriptions satisfy some simulation assumptions that make very easy their integration with checkers to perform dynamic ABV. On the contrary, ESW is generally far from guaranteeing the same assumptions, and in particular:

- *Timing references:* simulation of HW descriptions exploits either a cycle-based or an event-driven scheduling strategy which makes extremely clear how synchronization between modules happens, and exactly fixes the time when computation reaches a stable condition. Such timing reference provides designers with the exact time when checkers must be activated to verify the behavior of the design. The execution of ESW lacks such a timing reference, thus the checkers are not aware of the right time when ESW computation reaches a stable condition, nor they can control ESW concurrency.
- *Absence of complex data structures:* existing HW checker generators support the creation of assertion checkers according to the Boolean layer of PSL. Reasoning in terms of Boolean data type is straightforward in the HW domain, especially at RTL and gate level. On the contrary, ESW may involve more complex data structures, like

pointers, unions, etc., which cannot be easily handled by existing checker generators.

- *Variable visibility:* assertion checkers can monitor the behavior of variables which are made visible at the boundary of the module. For HW descriptions this means primary inputs and outputs and signals, which is enough to predicate about the functionality of the design. On the contrary, the scope of ESW variables is more complex and in some cases, relevant variables can be invisible to the checkers.

Due to such differences between HW and ESW, in a traditional (manual) ESW design approach, the integration between assertion checkers and the ESW under verification is not clear (Figure 1, right). Very often, the set up of a dynamic ABV environment requires an error-prone and tedious manual refinement of both the ESW code and the assertion checkers.

Despite of previous problems, some works, summarized in Section II, have been proposed to extend dynamic ABV towards ESW. However, to the best of our knowledge, none of them proposes a comprehensive solution that addresses all previous practical issues to provide ESW verification engineers with a framework for easy definition of assertions and their subsequent automatic synthesis into checkers suited for dynamic ABV of ESW.

This paper shows how such practical issues can be solved by exploiting the characteristics of *model-driven design* (MDD) [12] (Figure 1, left) based on synchronous models. Then, according to the proposed MDD-aware approach, we present an effective way for automatic generation of assertion checkers targeting ESW verification.

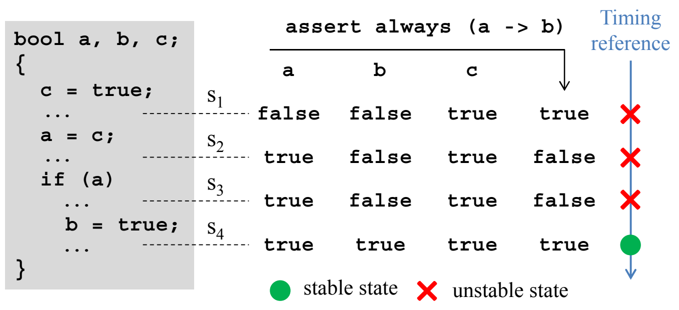
The remaining of the paper is organized as follows. Section II presents related works. Section III describes how model-driven design enables dynamic ABV of ESW. Section IV presents how to automatically synthesize checkers according to the MDD-based solutions described in Section III. Section V deals with experimental results and comparisons with other approaches. Finally, Section VI concerns concluding remarks.

II. Related Works

In literature, only few works propose approaches to apply dynamic ABV techniques to the ESW domain. In [13], the authors present a Microsoft-proprietary approach for binding C language with PSL. They define a subset of PSL and use a simulator as an execution platform. In this case, only a relative small set of temporal assertions can be defined, since only equality operator is supported for Boolean expressions, and the simulator limits the type of ESW applications.

Another extension of PSL is proposed in [14], where the authors unify assertion definition for HW and SW by translating their semantics to a common formal semantic basis. In [15], the authors use temporal expressions of *e* hardware verification language to define checkers. In both cases, temporal expressions are similar, but not compatible with PSL standards.

In [16] the authors propose two approaches based on SystemC. In the first case, ESW is executed on top of an emulated SystemC processor and, every clock cycle, the checkers monitor the variables and functions stored in the memory model. In the second approach, embedded software is translated in SystemC modules which run against checkers.



```
bool a, b, c;
{
  c = true;
  ...
  a = c;
  ...
  if (a)
    ...
    b = true;
    ...
}
```

assert always (a -> b)

	a	b	c	a -> b	
s_1	false	false	true	true	✗
s_2	true	false	true	false	✗
s_3	true	false	true	false	✗
s_4	true	true	true	true	●

Fig. 2: Is the assertion really violated in these states?

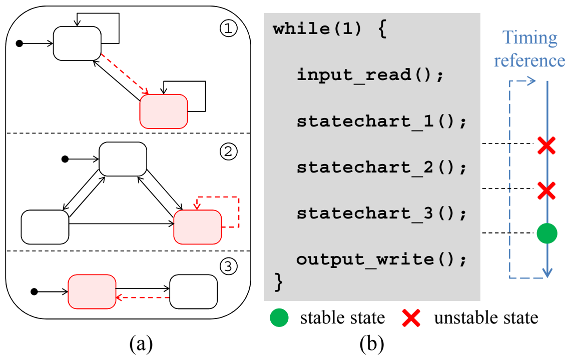
In this case, timing reference is imposed by introducing an event notified after each statement and the SystemC process is suspended on additional `wait()` statements. In both cases, there are some limitations. The SystemC (co-)simulation and the chosen timing references introduce significant overhead. In particular, clock cycle or statement step may be an excessive fine granularity for efficiently evaluating a sufficient number of assertions. Moreover, defining assertions relying on absolute time may generate significantly large checkers to address the high number of intermediate steps. Finally, it is difficult to define temporal assertions at source-code level, i.e., C applications, in terms of clock cycles.

III. Model-driven design for dynamic ABV of embedded software

Model-driven design refers to the use of graphical-modeling languages that allow to create an abstract model of the intended design. Such a model can, then, be simulated, verified and refined in an iterative manner, very often with the use of automatic tools, till specifications are satisfied. Then, the model is automatically synthesized into the final implementation for the target platform. As a drawback, the designer has limited control on the synthesis process: software-coding choices, e.g., functional partitioning and variable-scope assignment, are delegated to the synthesis tool, which has only to preserve the semantic imposed by the abstract model. However, from the point of view of dynamic ABV, what appears to be a drawback becomes an advantage. In fact, automatic synthesis of ESW prevents designers from implementing code that escapes from a pre-defined template. Thus, it enables the definition of a dynamic ABV approach that overcomes the practical issues summarized in the introduction. In particular, by using MDD, identification of timing references in ESW is extremely clear, complex data structures can be easily handled, and variable visibility, set at the time of the abstract model design, is preserved in the implemented code. Next paragraphs detail how dynamic ABV can be applied to ESW by exploiting the features of the radCASE industrial MDD tool [17]. Indeed, the same features are provided by several other MDD tools (e.g., [18], [19], [20]).

A. Timing references

Let us consider the piece of code and the assertion reported in Figure 2. By adopting the single statement as timing reference [16], the checker corresponding to the assertion is evaluated in states $s_1, \ldots, s_4$. For states s_2 and s_3, `a` is set to `true` and `b` not yet, thus the assertion is false, but does this *really* violate the intent of the verification engineer? Indeed, states $s_1, \ldots, s_3$ should be considered unstable, because the



```
while(1) {
   input_read();
   statechart_1();
   statechart_2();
   statechart_3();
   output_write();
}
```

Fig. 3: (a) Example of Statecharts. (b) Template of the corresponding synthesized code.

computation is still on-going. Unstable states characterize (embedded) software and may provide incorrect responses in terms of verification. Such unstable states are similar to the intermediate configurations, which characterize cycle-based or event-driven simulation of hardware descriptions. To overcome this issue, dynamic ABV in the HW evaluates temporal assertions when the system reaches a synchronization event, e.g., a clock event. Thus, by exploiting the timing reference which characterizes MDD approaches, we propose to adopt a similar strategy to identify the exact time when checkers must be activated to verify ESW.

In this work, we adopt the terminology model-driven design to refer specifically to ESW developed by using synchronous dataflow languages and models, which are similar to the ones provided with IAR VisualState [18], MathWorks Simulink [19] and Esterel Technologies SCADE Suite [20].

In a synchronous-modeling environment, components are designed as part of a single application: the synchronous model latches its inputs at the start of a computation step, computes the next system state and related outputs as a single atomic step, and communicates between components using dataflow signals. Signals consist of data values that are aligned with global clock ticks.

Such a synchronous model has affirmed for ESW design of safety critical avionics and automotive applications [21]. It differs from the more general class of modeling languages that includes support for asynchronous and concurrent execution of components and message passing [22]. The synchronous nature of such a model-driven class of ESW provides a native timing reference for triggering temporal assertions during dynamic ABV.

Let us consider Statecharts, i.e., a synchronous data-flow model, adopted by radCASE. They extend finite state machines with concepts of hierarchy, concurrency, and priority.

By using Statecharts, designers describe ESW functionalities in a concurrent, HW-like manner, then the MDD tool compiles away the concurrency. In particular, the synthesis process of radCASE represents Statecharts as a hierarchical switch-case construction, where each level of the state machine is a switch-case statement and the first-level machine is modeled as a separate function. In case of concurrency, the machines, i.e., function calls, are synthesized and ordered according to the priority specified by the designer. Figure 3(a) reports an example of a system composed of three modules. The simulation semantic of the synchronous model requires to traverse a transition at-a-time for each Statechart, i.e., the dashed transitions in the figure. When the last scheduled Statechart advances, the overall system moves in a new (*stable*) state, i.e., the filled states in the figure, and it is ready for the next step. radCASE synthesizes the Statecharts of the example as C-language functions scheduled and executed in a loop (Figure 3(b)): first, the inputs are read; then, each of the machines evolves; the *stable state* is reached after the last function returns; finally, outputs are written.

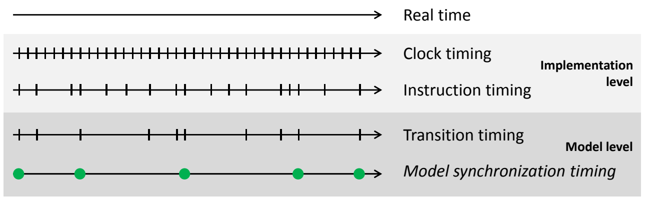


Fig. 4: Timing references for ESW.

In our proposal, checkers are activated only at the end of the loop of the model-driven ESW application, when each Statechart has executed at most one transition and the stable state is reached. Activation of checkers at a higher or lower rate may lead to their incorrect evaluation. Figure 4 shows different timing references for ESW. The meaning of clock and instruction timing is obvious, while we indicate, respectively, as transition and model synchronization timing, the instant when each single transition inside every Statecharts completes, and the instant when each Statechart has executed at most one transition and all are waiting for the new iteration of the main loop. Only model synchronization timing provides the right time for checker evaluation.

B. Variable visibility

C and C++ are the main used languages in the industry for the implementation of ESW applications [23]. Differently from HW description languages, e.g., VHDL, variables in C/C++ can be either of *global* or *local scope*. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is declared within a function body or a block. Global variables can be referred from anywhere in the code, vice versa the scope of local variables is limited to the block where they are declared.

This may compromise the dynamic ABV of ESW: verification engineers define assertions starting from system specifications and, in first instance, ignoring some of the implementation details; as well, designers build the system by refining the specifications in an actual implementation unaware of verification requirements. In particular, temporal assertions may predicate over both input/output parameters and internal variables. When assertions are synthesized in checkers, i.e., C functions, frequently some of the variables are out of the scope of the checker, i.e., not visible. This requires code refactoring, which is a time-consuming and error-prone activity (e.g., variable name clashing).

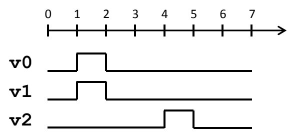
The integration of temporal assertion definition in the MDD environment easily addresses this issue, since each variable involved in the assertions can be automatically synthesized in the global scope of the application avoiding name clashing.

C. Complex data-structure

PSL, being an extension of LTL and CTL temporal logics towards HW design, natively supports the specification of

(a)

```
always (v0 & v1 → next[2] (v2))
```



(b)

```
always ((thermo.Heating = ON) &
        (thermo.Temperature >
          (thermo.Setpoint - thermo.Hysteresis)) ->
              (next[2] (thermo.Heating = OFF) )
```

Fig. 5: Example of a PSL assertion for HW (a) and ESW (b).

temporal assertions based on Boolean expressions. It supports also relational expressions on a restricted set of operands (integers, Booleans, bit vectors) which evaluates on a Boolean. Notice that RTL and gate-level HW descriptions can be easily brought down to the Boolean level, for example see Figure 5(a). On the contrary, temporal assertions for ESW may involve more complex conditional expressions, since C language is rich of data types, e.g., floats, pointers, structures, unions. But PSL does not support such data types. Figure 5(b) reports a temporal assertion defined for an ESW application. Its meaning is the same as in Figure 5(a), but the Boolean expression is built on top of more complex types and data-structures.

To take care of such a complexity, our checker generator synthesizes PSL formulas into checkers by adopting a *satellite*-based approach (details are in Section IV). A satellite is an arbitrary complex Boolean expression wrapped in a C function, that is not part of the checker temporal semantics.

Because of the adoption of a MDD environment, satellite definition can be automatically derived from assertions and ESW model. The ESW model stores the information about the variables data type, thus, how to retrieve the variable values (e.g., access to a `struct` field). This is necessary to formalize the assertion subformulas and store their values into Boolean variables that can be used as parameters for the checker evaluation function, as shown in Figure 6.

IV. Assertion checkers for embedded software

According to the ideas summarized in Section III, we implemented a framework for both assertion definition and checker generation. Describing how the semantics of an assertion is synthesized, i.e., the synthesis algorithm, is out of the scope of this paper (Section V shows that our checker generator is comparable with state-of-the-art checker generators for HW, concerning the support for PSL). Instead, we intend to highlight that our framework, by exploiting MDD features, handles variable visibility and complex data structures, and it supports exact identification of timing references for checker evaluation.

Variable visibility is guaranteed by the graphical editor that guides the *assertion definition*. A complete description of such a tool is out of the scope of this paper, since it has been already published in [24] for HW verification. Such an editor provides the user with all the design information necessary for completing parametric templates that will be automatically translated into PSL assertions. In the present work, we extend the editor to embedded software. In particular, what is important to emphasize is the role that the graphical editor plays

```c
void get_var(
    struct System const* sys,
    char const* name,
    void* var,
    long unsigned var_sizeof);

#define GET_VAR(T,N,S) \
  T N; get_var(sys, S, &N, sizeof(T));

void checker_wrapper(bool const enable,
    bool const reset,
    bool* fail,
    struct System const* sys) {

  // 1. Retrieve system-variable values
  GET_VAR(int8_t, thermo_Heating, "thermo.Heating");
  GET_VAR(int16_t, thermo_Hysteresis, "thermo.Hysteresis");
  GET_VAR(double, thermo_Setpoint, "thermo.Setpoint");
  GET_VAR(double, thermo_Temperature, "thermo.Temperature");

  // 2. Evaluate complex Boolean expressions
  bool v0 = (thermo_Heating == /* On */ 1);
  bool v1 = (thermo_Temperature >
      (thermo_Setpoint - thermo_Hysteresis));
  bool v2 = (thermo_Heating == /* OFF */ 0);

  // 3. Invoke the checker over Boolean variables
  checker_logic(enable, reset, fail, v0, v1, v2);
}
```

Fig. 6: The C-code of the satellite of the PSL formula in Figure 5.

in ensuring the right variable visibility to the designer during assertion definition. It analyzes the Statecharts representation, from which the ESW is synthesized, and it extracts structural information (i.e, module hierarchy, variable scope, etc.) to support the formalization of the ESW specifications into assertions. Thus, the tool is able to predict the functions definition that will be synthesized into the final ESW code. In this way, it provides the designer with all the formal parameters and variables organized by scopes. Then, variables referred by the designer into the assertions, are marked as visible within the Statechart model. As a consequence, these variables are synthesized in the ESW implementation in such a way they can be accessed by the assertion checkers. This prevents the designer from defining assertions involving variables that either will not be part of the ESW implementation or will not be accessible.

Support for *complex data structures* and identification of *timing references* are handled by the checker generator engine (CG). The CG synthesizes an assertion into compact procedural C-code functions that can be distinguished in: *logic*, *wrapper* and *checking* functions.

The *logic* functions (one for each assertion) implement the semantics of the checker. Logic function definition differs according to the assertion, but their signature is always characterized by the following set of Boolean parameters (Figure 7):

- the *enable* parameter is used to start the assertion evaluation in any instant of the simulation, and, in particular, to prevent a checker from scanning variables strictly before they have been initialized (i.e., in the unstable initial states);
- the *reset* parameter is used to (re-)initialize the checker to its initial configuration every time a new assertion evaluation has to be started;
- the *fail* parameter is set by the checker to notify an assertion violation;
- finally, the Boolean parameters $v_0 \ldots v_n$ represent the minimal subformulas of the assertion. The values of

```
//assert
// always (((thermo.Heating = On) &
//   (thermo.Temperature) >
//      (thermo.Setpoint - thermo.Hysteresis)) ->
//                  next[2] (thermo.Heating = Off))
// assert always v0 & v1 -> next[2] v2
#include <stdint.h>

typedef int bool;

void checker_logic(
    bool const enable,
    bool const reset,
    bool* fail,
    bool v0, bool v1, bool v2);
```

Fig. 7: The C-code signature of the logic function of the PSL formula in Figure 5.

```
1  typedef void (*checker_binding_func)(bool enable, bool reset,
2      bool* fail, struct System const* sys);
3
4  void checker_run_check(bool enable, bool reset,
5      struct System const* sys,
6      checker_binding_func* checkers) {
7    bool fail;
8    int unsigned i;
9    update_log();
10   for (i=0; checkers[i]; ++i) {
11    checkers[i](enable, reset, &fail, sys);
12    if (fail)
13      update_fail_log(i);
14   }
15 }
```

Fig. 8: The C-code of the checking function of the PSL formula in Figure 5.

these minimal subformulas are retrieved by the wrapper function associated to each logic function.

For lack of space, Figure 7 does not report the logic function implementation, but the handling of complex data structures is independent from it.

The *wrapper* function (Figure 6) contains the satellite expressions (lines `22-25`) that handle the use of ESW complex data structures inside PSL assertions. These expressions act as actual parameters to call the corresponding logic function (line `28`). The satellite expressions implement the minimal subformulas of a PSL assertion (e.g., `thermo.Heating = On`, `thermo.Temperature > (thermo.Setpoint - thermo.Hysteresis)`, `thermo.Heating = Off`). They are automatically extracted from the data dump of the corresponding PSL assertion created by the assertion editor. Notice that, the data dump stores the name of the ESW variables involved into satellite expressions, as well as the variables type. In this way, the CG correctly retrieves, from a structure containing all the global variables defined in the ESW (i.e., `sys`, line `2`), the value of variables (lines `16-19`) used to evaluate the satellite expressions.

The *checking* function (Figure 8) is used to perform a verification step when a stable configuration of the ESW is reached. It executes all the synthesized checkers by calling the corresponding wrapper functions (e.g., `checker_wrapper`, line `11`). Moreover, it manages a log file in which it stores the simulation traces (i.e., variable evolution). At each invocation, labeled with a time stamp inside the log file, the current values of variables are recorded (line `9`) and, error notification is reported whenever an assertion violation occurs(lines `12-13`). In this way, simulation traces can be used for debugging purposes. In fact, the trace portion that precedes an error message into the log file represents a counterexample to the satisfiability of the assertion.

V. Experimental Analysis

The proposed dynamic ABV methodology for ESW has been evaluated from two points of view: (i) efficiency and effectiveness, and (ii) quality of the checker generation.

A. ABV efficiency and effectiveness

The proposed approach has been implemented in radCHECK [25], as an extension of the radCASE virtual prototyping simulator, by integrating the assertion editor and the checker generator.

TABLE I: Case-studies characteristics.

Design	Specs	Modules	LoC	Asserts
DSC	25	93	26000	211
Tridomix	12	18	10782	163
Desal	10	28	5319	113
HV	7	18	2626	88
Sewing	40	164	120638	258

TABLE II: ABV efficiency and effectiveness.

		MSync		Trans		Instr	
Design	Asserts	FN/P	Time	FN/P	Time	FN/P	Time
DSC	211	0	50	27	2088	184	99105
Tridomix	163	0	36	42	327	139	11759
Desal	113	0	17	16	337	102	13294
HV	88	0	11	29	16.8	72	2358
Sewing	258	0	161	52	4577	213	226146

Time is expressed in seconds.

Table I reports the characteristics of some industrial ESW for control and automation systems, that have been developed with radCASE. In particular, column *Design* reports the design name; *Specs* is the number of pages in the specification and requirement document (in natural language); *Modules* is the number of the Statecharts modules in the radCASE model; *LoC* is line of code of the synthesized ESW; finally, *Asserts* is the number of assertions which engineers defined for the ESW verification.

We have compared the efficiency and effectiveness of the proposed dynamic ABV for embedded software with respect to different timing references. In particular, we measure the efficiency in terms of execution time of the ESW running with checkers; we evaluate the effectiveness of the adopted timing reference by counting the number of false negatives and false positives which arise during the checker simulations. A false negative (positive) occurs when the checker indicates that an assertion does not hold (hold) when it really does (does not).

Table II reports such an analysis. Column *Design* and *Asserts* report the design name and number of associated assertions, respectively; column *MSync* reports the results of our approach, which evaluates checkers according to the model synchronization timing; *Trans* refers to the transition timing; finally, *Instr* refers to instruction timing, as described in [16]. For each approach, *FN/P* is the number of false negatives or false positives which occur, and *Time* is the execution time in seconds. We have simulated each design in the radCASE/radCHECK environment with the respective test suites containing approximately 10^5 test vectors.

As expected, since all designs meet the specifications, we do not have false negatives/positives in case of using the model synchronization timing reference. Vice versa, we can observe that false negatives/positives occur adopting different timing reference: typically this is due to assertions that predicate over variables before a stable condition is reached (e.g. Figure 2). In particular, the number of false negatives/positives is greater when assertions are evaluated according to the instruction timing. Whereas, in case of transition timing, only assertions whose variables span over different modules may be affected by false negatives/positives.

Finally, increasing the checker-evaluation rate significantly impacts the overall execution time, as columns *Time* show.

B. Checker generation

The characteristics of our checker generation engine have been compared with two state-of-the-art checker generators for HW. In particular, Table III reports the set of PSL operators supported by our tool (CG) w.r.t. FoCs [10] and MBAC [4].

TABLE III: Set of supported PSL operators.

PSL operator	**CG**	**FoCs**	**MBAC**
next, **next_a**, **next_e** *family*	✓	✓	✓
next!, **next_a!**, **next_e!** *family*	✓	-	✓
next_event *family*	✓	✓	✓
next_event! *family*	✓	-	✓
next_event_a *family*	✓	-	✓
next_event_a! *family*	✓	-	✓
next_event_e *family*	✓	-	✓
next_event_e! *family*	✓	-	✓
before *family*	✓	✓	✓
before! *family*	✓	-	✓
until *family*	✓	✓	✓
until! *family*	✓	✓	✓
eventually!	✓	✓	✓
always	✓	✓	✓
logical *FL operators*	✓	✓	✓
SERE suffix implication	✓	✓	✓
SERE consecutive repetition	✓	✓	✓
SERE non-consecutive repetition	✓	✓	✓
SERE goto repetition	✓	✓	✓
SERE or	✓*	✓*	✓
SERE non-length-matching and	✓*	✓*	✓
SERE length-matching and	✓*	✓*	✓
SERE within	✓	-	✓

* not supported on the RHS of the suffix implication operator

It is worth noting that our CG supports more operators w.r.t. FoCs but, unlike MBAC, it introduces some restrictions on the SERE operators. This is due to the fact that the generation of a procedural code implementation of the “or”, the “length-matching and” and the “non-length matching and” operators on the right-hand-side (RHS) of a suffix implication requires further investigation for guaranteeing the correctness of the checker. On the contrary, the checker implementation generated by MBAC seems to not suffer of this problem. However, MBAC has been explicitly developed for supporting the generation of checker in the hardware-context, while, our CG generates C-code checkers to be used for dynamic ABV of ESW.

VI. Concluding Remarks

The paper showed how model-driven design enables dynamic ABV for embedded SW. Differences between HW descriptions and ESW applications, which prevent an easy porting of HW-oriented ABV methodologies towards ESW, have been discussed. Then, MDD-based solutions have been proposed to solve practical issues concerning integration of assertion checkers in ESW simulation. In particular, we showed that following an MDD approach allows to identify the right instants for checker evaluation, as well as it provides support for handling of variable visibility and complex data-structure inside ESW code. The proposed solutions have been implemented in a dynamic ABV framework composed of an assertion editor and a checker generator which has been integrated in an industrial MDD environment for control and automation ESW. Experimental results highlighted the effectiveness and the efficiency of the proposed solutions.

References

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-based Design*. Springer Netherlands, 2004.

[2] M. Kim, Y. Kim, and H. Kim, “A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study,” *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 146–160, 2011.

[3] “IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2010),” IEEE Computer Society, pp. 1–171, 2010.

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

[5] L. Ferro and L. Pierre, “ISIS: Runtime Verification of TLM Platforms,” *Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs*, vol. 63, pp. 213–226, 2010.

[6] Cadence, “Assertion-based Verification.” [Online]. Available: http://www.cadence.com/products/fv/pages/abv_flow.aspx

[7] Mentor Graphics, “Assertion-based Verification.” [Online]. Available: http://www.mentor.com/products/fv/methodologies/abv/

[8] S. Shukla, A. Hu, J. Abrahams, P. Ashar, H. Foster, A. Landver, and C. Pixley, “Panel: Assertion-Based Verification-What’s the Big Deal?” in *Proc. of IEEE International High Level Design Validation and Test Workshop*, 2006.

[9] H. Foster, K. Larsen, and M. Turpin, “Introducing the New Accellera Open Verification Library Standard,” in *Proc. of Design and Verification Conference*, 2006.

[10] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal, “FoCs - Automatic Generation of Simulation Checkers from Formal Specifications,” in *Computer Aided Verification*, ser. Lecture Notes in Computer Science. Springer Berlin, 2000, vol. 1855, pp. 538–542.

[11] D. Tabakov and M. Vardi, “Optimized Temporal Monitors for SystemC,” in *Proc. of International Conference on Runtime Verification (RV)*. Springer, 2010, pp. 436–451.

[12] B. Selic, “The Pragmatics of Model-driven Development,” *Software, IEEE*, vol. 20, no. 5, pp. 19–25, 2003.

[13] P. Cheung and A. Forin, “A C-language binding for PSL,” *Embedded Software and Systems*, pp. 584–591, 2007.

[14] F. Xie and H. Liu, “Unified Property Specification for Hardware/Software Co-verification,” in *Proc. of International Computer Software and Applications Conference*. IEEE Computer Society, 2007, pp. 483–490.

[15] M. Winterholer, “Transaction-based Hardware Software Co-verification,” in *In Proc. of Forum on Specification & Design Languages*, 2006.

[16] D. Lettnin, P. Nalla, J. Ruf, T. Kropf, W. Rosenstiel, T. Kirsten, V. Schonknecht, and S. Reitemeyer, “Verification of Temporal Properties in Automotive Embedded Software,” in *Proc. of Design, Automation, and Test in Europe*. ACM, 2008, pp. 164–169.

[17] STMProducts, IMACS, “radCASE.” [Online]. Available: http://www.stm-case.com/en/index.php

[18] IAR Systems, “IAR visualSTATE.” [Online]. Available: http://www.iar.com/website1/1.0.1.0/371/1/

[19] The MathWorks, “Simulink.” [Online]. Available: http://www.mathworks.com/products/simulink

[20] E. Technologies, “Scade suite.” [Online]. Available: http://www.esterel-technologies.com/products/scade-suite

[21] M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni-Vincentelli, “Efficient Embedded Software Design with Synchronous Models,” in *Proc. of ACM International Conference on Embedded Software*, 2005, pp. 187–190.

[22] A. Benveniste and G. Berry, “The Synchronous Approach to Reactive and Real-time Systems,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270–1282, 1991.

[23] D. Lewis, *Fundamentals of Embedded Software*. Prentice-Hall, 2002.

[24] L. Di Guglielmo, F. Fummi, N. Orlandi, and G. Pravadelli, “DDPSL: An Easy Way of Defining Properties,” in *In Proc. of IEEE International Conference on Computer Design*, 2010, pp. 468–473.

[25] STM Products s.r.l., “radCHECK.” [Online]. Available: http://www.verificationsuite.com