Formal Verification of Embedded Systems's Components at the System-Level

Mateus Krepsky Ludwich, Antônio Augusto Fröhlich
Laboratory for Software e Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
P.O.Box 476, 880400900 - Florianópolis - SC - Brasil
Email: {mateus,guto}@lisha.ufsc.br

***Abstract*—The increasing complexity of embedded systems is pushing their design to the System-Level, leading to the convergence between software and hardware. In such scenario, it is highly desirable to formally verify properties of such systems regardless of whether their components are going to be implemented in software or hardware. In this paper we introduce an approach to formally verify functional correctness properties and safety properties of embedded systems components at System-Level. In order to demonstrate our approach, we present a scheduler of an embedded operating system showing that such scheduler follows its specification regardless of the domain it is instantiated.**

I. Introduction

The increasing complexity of embedded systems is pushing their design to higher levels of abstraction, leading to the development of methodologies that are known to work at the *System-Level*, where there is no distinction between software and hardware [1], [2]. In such scenario, it is highly desirable to formally verify properties of such systems regardless of whether their components are going to be implemented in software or hardware.

Two important properties target of formal verification are: functional correctness and safety. Functional correctness aims to check if a given implementation follows its specification (also referred as its *contract*). Safety aims to check if there is a path of execution which leads the component to an error state. An error state can be caused, for example, by buffer overflows (e.g. while array bounds are surpassed), and by violating pointer safety (e.g. while dereferencing a null pointer).

In this paper we introduce an approach to formally verify functional correctness properties and safety properties of embedded systems components at System-Level. In our proposal, the contract is composed by classes invariants and methods preconditions and postconditions as it is proposed by *contract programming* [3]. Such contract is specified in C++, the same language used for the implementation. Both specification and implementation are translated to the internal representation of the C Bounded Model Checker (CBMC) [4] and then, formally checked. Besides verifying functional correctness properties, specified by the components' contracts, CBMC also checks for safety properties such as the absence of buffer overflows, and pointer safety.

In order to demonstrate our approach, we present a scheduler of an embedded operating system showing that such scheduler follows its specification regardless it is instantiated in the software or in the hardware domain.

The remaining of this paper is organized as follows: Section II makes an overview of formal verification techniques for System Level Design (SLD), and formal verification of embedded systems components; Section III presents our approach for System-Level verification; Section IV evaluates our approach for the scheduler of an embedded operating system; Section V closes the paper with our final considerations.

II. Related Works

This section overviews the main languages and techniques used for formal verification of System-Level descriptions, and works related to formal verification of embedded and operating systems component's.

In the last few years, advances in Electronic Design Automation (EDA) techniques and tools are allowing hardware synthesis from high-level behavioral models. This process, known as High-level Synthesis (HLS), allows designers to describe hardware components using programming languages such as C++ and Java, and System-Level Description Languages (SLDLs) such as SystemC and SpecC [1].

As HLS allows for describing hardware components using programming languages, such descriptions can be verified using tools based on Software Model Checking. In Software Model Checking a program is translated into a logical formula. Properties about the target program can be described in such logics, directly in the programming language (e.g. by using *assert* expressions), or even automatically generated by analyzing constructions of the programming language, such as arrays and pointers. Then, the logical formula representing the program is combined to such properties, generating *Verification Conditions* (VCCs) that are passed to a satisfiability (SAT) solver or to a theorem prover. Finally, it is determined if all properties of the program are true or not. In the latter case, it is generated a counter-example, demonstrating the execution path where the property is false. Are examples of tools that use software model checking concepts for checking C/C++ programs: Blast (C only) [5], CBMC (used in this work) [4], LLBMC [6], and SATABS [7].

Model checking and other verification approach also have been applied to SLDLs such as SystemC and Spec. One of the approaches used is to transform SLDLs in C++ and

then use software model checking tools. That is the case of KRATOS model checker [8], and the Scoot tool [9], both targeting SystemC descriptions. Similar strategy is used by Clarke for verifying SpecC descriptions [10]. Besides C++ other languages are used as intermediate to model checking SLDLs descriptions. For example, the Afra tool [11], translates SystemC descriptions to the Rebeca modeling language that can be latter checked by the Modere model checkers which uses Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) to specify properties target of verification.

Using the ideas aforementioned, Fujita presents the verification of synchronization properties for the Point-to-Point Protocol (PPP) [12]. In Fujita's work, the SpecC description of PPP is translated into Boolean SpecC and then model checking techniques are employed. On the operating system field, Klein presents the results of the L4.Verified project, which accomplished the verification of the seL4 micro-kernel used in embedded systems [13]. The strategy used by the L4.Verified project was to translate the C implementation of seL4 operating system into the Simpl language [14] implemented using High-Order Logic (HOL) on the Isabelle theorem prover [15]. Then, using Isabelle, invariants about the representation of the operating system components could be proved. The L4.Verified project also had proved, using refinement, that such Simpl description of the components were equivalent to higher level descriptions, also described using HOL. Similarly to the L4.Verified project, Cohen presents the verification of functional correctness of the Microsoft's Hyper-V virtualization platform and of SYSGO's embedded real-time operating system PikeOS [16]. Cohen's approach proposes a logics to express Locally Checked Invariants (LCI). LCI are used to assign to a component target of verification a predicate over pairs of states expected to hold that are then checked using the VCC theorem prover [17]. Gotstman proposes the modular verification of preemptive kernels, running in a multiprocessor system [18]. Gotsman's approach propose two proof systems, one called high-level where all process have their own virtual CPU and other called low-level where there are a fixed number of processors. The high-level proof system is then used to reason about the schedulable code, and the lower-level proof to reason about the scheduling itself, separating the scheduler from the rest of the kernel. Although the proposal is to verify mainstreams operating systems such as Linux, FreeBSD and XNU, all theory presented in the paper is developed around an assembly language of a fictional machine.

At this work, we propose the verification of functional correctness properties and safety properties of components described at the system-level. Differently from [13] and [16], the language used to express the component's contract is the same used for the component's implementation (C++). Therefore, there is no need to proof refinement from higher levels of the specification. Also, because we choose using a programming language to specify contracts, no special knowledge on formal logics is required from the designers in order to express the component's expected behavior.

III. Proposal

As pointed out in Section II, recent progress in High Level Synthesis tools and techniques, are enabling the generation of hardware components from C++ descriptions. Furthermore, C++ has been used as a mainstream language for developing embedded software. Therefore, we have chosen the C++ language to describe embedded systems components which are going to be subject of formal verification. However, as we explain in Section V, the general ideas of our approach can be applied in others System Level Design Languages (SLDL), such as SystemC and SpecC.

Our proposed approach for formal verification of embedded systems components can be divided in three main stages: first one writes the contract of the component to be verified, then one instrument the component implementation with such contract, finally one executes software bounded model checking to verify if the component implementation respects its contract. The next paragraphs of this section detail such process.

According to our approach, the component contract is written in C++ using assertions (*assert* expressions). A contract is composed by the invariants of the class that defines the component, and by a set of preconditions and postconditions (one for each public method). Class invariants are defined as a set of assertions that are always true for all instances of a class. The preconditions of a method are defined as a set of assertions that need to be true in order to the method execute. Similarly, the postconditions of a method are defined as a set of assertions that need to be true in case the method terminates normally. The assertions of the preconditions and postconditions can reason about the method's parameter, return value, and object state. Figure 1 shows an example of contract for the *insert* method of a queue class. The *invariants* method implements the class invariants, which can contain something like

```
assert(size() >= 1);
```

meaning that such queue is never empty.

The main disadvantage of writing contracts directly on the component class, as represented by Figure 1, is the additional code that the contract adds in to the component. While such code is needed during the verification phase, its presence might be undesired while the component is been used in a running system, since such additional code means a higher resource usage (such as memory and energy), maybe demanding more that the original embedded system was projected for. While the assertion expressions can be disabled if implemented using macros from the C preprocessor, as is the case of the standard C library (assert.h file for C or cassert file for C++), some other more complex contracts cannot. That is the case, for example, of the expression

```
assert(size() == size_at_pre + 1);
```

of Figure 1, since it depends on the definition of a new variable (*size_at_pre*) not present in the original method, but needed to express the method's contract.

```
void insert(int obj)
{
  // preconditions
  unsigned int size_at_pre = size();
  assert(! contains(obj));

  // Class invariants
  invariants ();

  // Method's implementation
  // ...

  // Class invariants
  invariants ();

  // postconditions
  assert(size() == size_at_pre + 1);
  assert(contains(obj));
}
```

Figure 1. Example of contract

As an alternative for writing contracts directly on the component class, one can face the instrumentation of a component for verification as placing such component in a scenario for verification, which we named the *Verified Scenario*, shown by Figure 2. Verified Scenario is implemented using the *Scenario Adapter* pattern [19]. The client (*Client*) class of a component access its methods though the scenario adapter (*ScenarioAdapter*), which extends the desired implementation of a certain component. For each public method of the component, the *ScenarioAdapter*, applies the pattern shown at the implementation of the method *operation*. It calls the preconditions of such method before calling the actual method, and calls postconditions after the actual method is called. The actual method is called through delegation (*ComponentImp::operation()*). It also calls the *invariants* just before and just after calling the actual method, in order to guarantee that the method's implementation does not violate the class invariants. The *VerifiedScenario*, which is composed together with the *ScenarioAdapter* through inheritance, is the one that contains all methods preconditions and postconditions and the class invariants. Such preconditions, postconditions and invariants can be faced as *aspects* that change the behavior of a method making it "verifiable". The *VerifiedScenario* groups such *contract aspects* together also using inheritance.

The main advantage of using the scenario adapter pattern to instrument a component with its contract is the possibility of enabling and disabling such instrumentation easily. In order to enable the instrumentation all it is necessary to do is to create a type alias.

```
typedef ScenarioAdapter<ComponentImp>
ComponentImp;
```

As the scenario adapter implements the public interface of the *Component* interface, the client class accesses the verified version of the component as usually.

The verification proposed in this work is made in an offline form (i.e. before the component is on a running system)

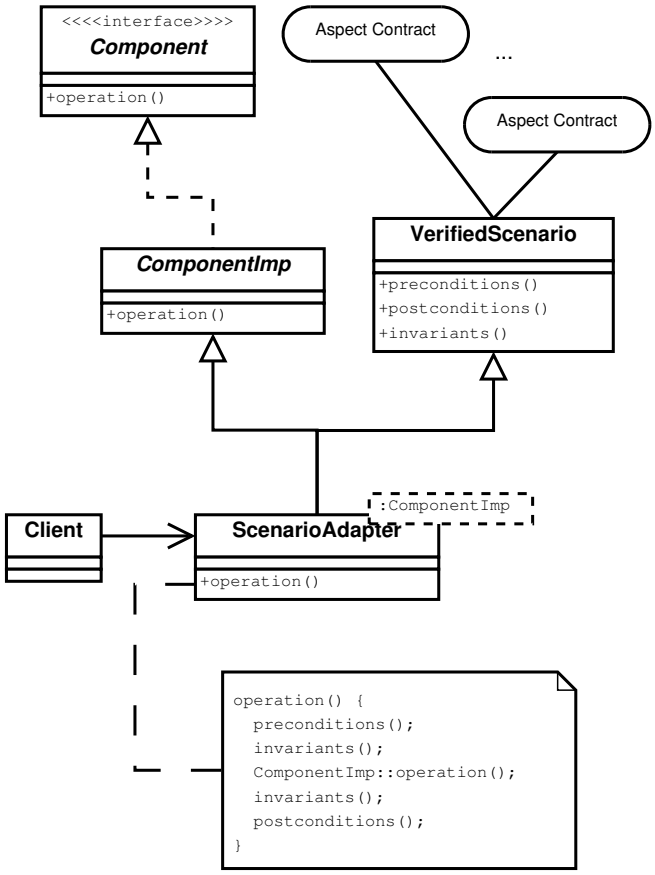

Figure 2. Verified Scenario

by performing software model checking on the C++ code. Therefore, the running version of the component does not need of its contract anymore (since it was already proved on the verification phase that the component follows its contract). However, one can choose the keep the instrumented version of the component at the runtime to reuse its contract, for example, for monitoring proposes or for unit testing. This is feasible as long the additional code generated by the component's contract is taken into consideration while designing the embedded system.

The tool we have chosen to perform software model checking on embedded system components was the *C Bounded Model Checker* (CBMC) [4]. The component implementation, already instrumented with its contract, is translated by CBMC into a logical formula by a process called *symbolic simulation*. While performing this translation, CBMC also perform static analysis on the source code generating properties to be checked. Among this properties are the user defined assertions which, in our case, compose the component's contract and safety properties such as array bound check and pointer safety check. Then, the logical formula representing the component is combined to the properties generated by CBMC, generating VCCs. Finally, CBMC invokes an SAT solver which determines if the component properties are true (i.e. if they *hold*). In the case a properties does not hold, is generated a counter-

example showing the execution path where the property is false. If all properties are proved true the verification ends successfully, proving that the component fulfills its contract and is free of safety properties violations.

IV. Case study

We have used the scheduler of the Embedded Parallel Operating System (EPOS) [20] as a case study for our proposal, showing that it follows its formal contract. As shows Figure 3, in the design of the EPOS's scheduler queue management is independent from scheduling policy [21]. Furthermore, the scheduler itself is independent from the object it schedules. In the case of Figure 3, the schedulable object is an operating system thread. Schedulable objects and the scheduler are united by the *scheduling criteria*, which defines a scheduling policy algorithm. The scheduling criteria defines and *int()* operator, which maps a schedule object into a integer allowing for schedulable objects (e.g. threads) to be ordered in the scheduler queue using ordinary integer operands of comparison (e.g. $\leq$, $\neq$).

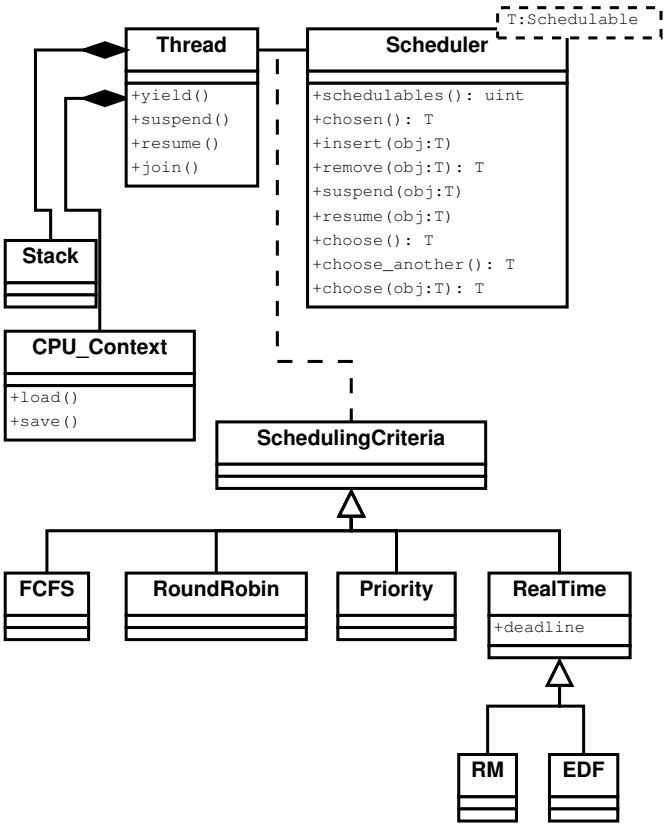

Figure 3. EPOS's scheduler

In this paper we focus on the verification of the scheduler alone, not depending on what is scheduled. We have verified the C++ implementation of the EPOS's Scheduler. Such implementation can be compiled using GCC in order to generate the software version of the scheduler or can be synthesized using HLS tools in order to generate the hardware version of the scheduler [22].

We have verified functional correctness properties and safety properties of the EPOS's scheduler. The functional correctness properties are specified by the scheduler's contract, which is composed by the class invariant

```
assert(Base::size() >= 0);
```

meaning that the *ready* queue could never have a negative number of schedulable objects, and by the preconditions and postconditions of each public method of the *Scheduler* class, shown at the Table I. The assertions are presented in C++ syntax as they are implemented. Some variable assignments are omitted for the sake of the presentation. Variables using the suffix "at_pre" contain the value obtaining while the method's preconditions were checked (and before the execution of the method's body). The protected method *contains* checks whether the *ready* queue contains the schedulable object passed as parameter. The safety properties of the scheduler contract are related to array bounds and pointer safety and are automatically generated by CBMC.

We have instrumented the scheduler for verification using the *Verified Scenario* as described in Section III. Then we have used CBMC to check the scheduler implementation already instrumented with its contract. CBMC has generated *M* VCCs related to functional correctness properties, *N* VCCs related to array bounds and *O* VCCs related to pointer safety. All VCCs hold, proving the scheduler correctness and safety properties.

V. Conclusion

In this paper, we have introduced an approach to formally verify functional correctness properties and safety properties (the absence of buffer overflows and pointer safety) of embedded system components described at System-Level. The functional correctness properties of a component are expressed by its contract containing class invariants, and preconditions and postconditions for each public method. The safety properties are automatically generated by the CBMC model checker, used in this work. The component target of verification is instrumented with its contract using the *Verified Scenario*, and the resulting component is verified by CBMC.

We have chosen the C++ language for implementing the component and for specifying its contract thus, no special knowledge about formal methods is required from the component designer.

Although we have chosen C++ and CBMC we believe that the general ideas of our approach can easily be adapted for other languages and model checkers. In the case of using SystemC, the verified scenario can be directly applied, since SystemC is implemented as a C++ library. In the case of SpecC the verified scenario can be applied by using aggregation instead of inheritance in order to compose the verification aspects with the *VerifiedScenario*, and the *VerifiedScenario* with the *ScenarioAdapter*. Then, any model checker supporting such languages and supporting user-defined assertions can be used.

In order to evaluate out approach we have formally verified the scheduler component of the EPOS embedded operating

Table I
SCHEDULER'S CONTRACT

Method	Preconditions	Postconditions
unsigned int schedulables()	None	None
T volatile chosen()*	None	**assert**(contains(var_chosen));
void insert(T obj)*	**assert**(! contains(obj))	**assert**(contains(obj)); **assert**(Base::size() == size_at_pre + 1);
T remove(T* obj)*	None	**assert**(! contains(obj)); **if**(contains_obj_at_pre) **assert**(Base::size() == size_at_pre - 1); **else** **assert**(result == null);
void suspend(T obj)*	None	**assert**(! contains(obj)); **if**(contains_obj_at_pre) **assert**(Base::size() == size_at_pre - 1);
*void resume(T * obj)*	*same as insert*	*same as insert*
T choose()*	None	**assert**(contains(var_chosen)); **assert**(Base::size() == size_at_pre);
T choose_another()*	None	**assert**(contains(var_another)); **assert**(Base::size() == size_at_pre);
T choose(T* obj)*	None	**assert**(Base::size() == size_at_pre); **if**(contains_obj_at_pre) **assert**(contains(obj)); **else** **assert**(result == null);

system. A software version of such scheduler can be obtained using the GCC compiler and, using HLS tools, a hardware version can be synthesized from the same scheduler description.

In this paper we have presented the verification of the scheduler independently of the object been schedule. As future directions for this work we planning on verifying thread abstractions of the operating system thus, verifying the whole task scheduling of the system.

REFERENCES

[1] A. Gerstlauer, C. Haubelt, A. D. Pimentel, T. P. Stefanov, D. D. Gajski, and J. Teich, "Electronic system-level synthesis methodologies," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 10, pp. 1517–1530, Oct. 2009.

[2] P. R. Panda, "SystemC: a modeling platform supporting multiple design abstractions," in *Proc. of the 14th international symposium on Systems synthesis*, Montreal, Canada, 2001, pp. 75–80.

[3] B. Meyer, "A framework for proving contract-equipped classes," in *Proceedings of the abstract state machines 10th international conference on Advances in theory and practice*, ser. ASM'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 108–125.

[4] D. Kröning, "The cbmc homepage," 2012. [Online]. Available: http://www.cprover.org/cbmc/

[5] T. A. Henzinger, R. Majumdar, D. Beyer, R. Jhala, G. Sutre, and A. Chlipala, "Blast: Berkeley lazy abstraction software verification tool," 2012. [Online]. Available: http://www.cs.ucla.edu/ rupak/blast/

[6] C. Sinz, S. Falke, and F. Merz, "Llbmc: The low-level bounded model checker," 2012. [Online]. Available: http://llbmc.org/

[7] D. Kroening and E. Clarke, "Satabs - predicate abstraction using sat," 2012. [Online]. Available: http://www.cprover.org/satabs/

[8] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos: a software model checker for systemc," in *Proceedings of the 23rd international conference on Computer aided verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 310–316.

[9] N. Blanc, D. Kroening, and N. Sharygina, "Scoot: a tool for the analysis of systemc models," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 467–470.

[10] E. Clarke, H. Jain, and D. Kroening, "Verification of specc using predicate abstraction," *Form. Methods Syst. Des.*, vol. 30, no. 1, pp. 5–28, Feb. 2007.

[11] N. Razavi, R. Behjati, H. Sabouri, E. Khamespanah, A. Shali, and M. Sirjani, "Sysfier: Actor-based formal verification of systemc," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 19:1–19:35, Jan. 2011.

[12] M. Fujita, I. Ghosh, and M. Prasad, *Verification Techniques for System-Level Design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[13] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "sel4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220.

[14] N. Schirmer, "Verification of sequential imperative programs in Isabelle/HOL," Ph.D. dissertation, Technische Universität München, 2006.

[15] T. Nipkow, M. Wenzel, and L. C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*. Berlin, Heidelberg: Springer-Verlag, 2002.

[16] E. Cohen, M. Moskal, W. Schulte, and S. Tobies, "Local verification of global invariants in concurrent programs," in *Proceedings of the 22nd international conference on Computer Aided Verification*, ser. CAV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 480–494.

[17] M. Corporation, "Vcc mechanical verifier for concurrent c programs," 2012. [Online]. Available: http://vcc.codeplex.com/

[18] A. Gotsman and H. Yang, "Modular verification of preemptive os kernels," *SIGPLAN Not.*, vol. 46, no. 9, pp. 404–417, Sep. 2011.

[19] A. A. Fröhlich and W. Schröder-Preikschat, "Scenario adapters: Efficiently adapting components," Orlando, USA, 2000.

[20] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[21] H. Marcondes, R. Cancian, M. Stemmer, and A. A. Fröhlich, "On design of flexible real time schedulers for embedded systems," in *International Symposium on Embedded and Pervasive Systems*, Vancouver, Canada, Aug. 2009, pp. 382–387.

[22] J. Flor, T. Muck, and A. Frohlich, "High-level design and synthesis of a resource scheduler," in *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, dec. 2011, pp. 736 –739.