On the Formal Verification of Component-based Embedded Operating Systems

Mateus Krepsky Ludwich and Antônio Augusto Fröhlich
Federal University of Santa Catarina (UFSC)
Software/Hardware Integration Lab (LISHA)
880400-900 - Florianópolis - SC - Brazil
{mateus,guto}@lisha.ufsc.br

ABSTRACT

The increasing complexity of embedded systems is pushing their design to System-Level, thus leading to a convergence between software and hardware. Consequently, operating systems in this realm are also being required to deliver their services both as software and as hardware. In such a scenario, it is desirable to verify system properties regardless of whether its components are instantiated at software or hardware. In this paper, we describe an approach to formally verify functional correctness and safety properties of such system-level component. The approach is illustrated by a case study of EPOS' scheduler, whose implementation can be driven to yield both a software instance compiled by the GCC C++ compiler or a hardware instance synthesized by the CatapultC ESL tool. We demonstrate that the scheduler follows its specification regardless of the domain for which it is instantiated. We also demonstrate that the proposed approach causes no run-time overhead, since the adopted Software Model Checking techniques are deployed at compile-time.

Categories and Subject Descriptors

D.2.4 [**Program Verification**]: Correctness proofs; D.4.1 [**Process Management**]: Scheduling; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Algorithms, Design, Verification

Keywords

Formal Verification, Embedded Operating Systems, Scheduling

1. INTRODUCTION

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 [12]. In such scenario, it is highly desirable to verify system properties formally, regardless of whether their components are going to be implemented in software or hardware.

Two main classes of properties for formal verification are functional correctness and safety. Functional correctness aims to check if a given implementation follows its functional specification (also referred as *contract*). Safety aims to check if there is an execution path 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).

As HLS allows for hardware components to be described using programming languages, descriptions can now be verified using Software Model Checking tools. 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 with given properties, generating *Verification Conditions* (VCCs) that are passed to a satisfiability solver or to a theorem prover. Finally, it is determined if all properties specified for the program are true or not. In the latter case, a counter-example is generated, demonstrating the execution path that lead the property to be false.

Model checking and other verification approaches also have been applied to SLDLs such as SystemC and SpecC. Usually, SLDL descriptions are transformed in C++ code that is subsequently submitted to a software model checking tool. That is the case of the KRATOS model checker [3] and the Scoot tool [1], both targeting SystemC descriptions. A similar strategy is used by Clarke to verify SpecC descriptions [4]. However, recent Electronic System-Level (ESL) enable hardware components to be directly described in C++. Certainly some issues are intrinsic to the design of software and others to hardware, but in respect to formal verification, this new generation of tools allow us to skip model transformations and language translations as long as components are properly designed for verifiability.

In this paper, we propose an approach to formally ver-

SBESC 2012 Natal, Brazil

ify functional correctness and safety properties of embedded system components described at system-level. In our proposal, a contract is written for each component in order to formally specify its behavior. Contracts are composed by class invariants and method pre and postconditions as proposed by Meyer in *Design by Contract* [17]. Such contracts are also specified in C++, the same language used for the implementation of components. Both specification and implementation are translated to the internal representation of the C Bounded Model Checker (CBMC) [15] and then formally checked. Besides verifying functional correctness properties specified by a component's contract, CBMC also checks for safety properties such as the absence of buffer overflows, and pointer safety. In order to demonstrate our approach, we have verified the scheduler of the Embedded Parallel Operating System (EPOS) [9] showing that it follows its specification regardless of being instantiated as software or hardware.

The remaining of this paper is organized as follows: Section 2 makes an overview of formal verification at System Level Design (SLD), and formal verification of embedded systems; Section 3 presents our approach for System-Level verification; Section 4 evaluates our approach for the scheduler of an embedded operating system; Section 5 closes the paper with our final considerations.

2. RELATED WORKS

At the System-Level, Fujita and others have described the verification of synchronization properties for the Point-to-Point Protocol (PPP) [11]. In their work, the SpecC description of PPP is translated into Boolean SpecC and then translated into mathematical representations of equalities and inequalities that are solved by an Integer Linear Programming (ILP) solver. Computation Tree Logic (CTL) formulas are used to represent the expected protocol behavior. The protocol implementation is checked for state reachability and deadlock conditions. The adoption of ILP techniques has the advantage of yielding a model that could be used for system optimization in addition to verification, but at the same time, imposes practical limits to what can be verified in this manner.

On the operating system realm, Klein and others have presented the results of the L4.Verified project, which accomplished the verification of the seL4 μ-kernel used in embedded systems [14]. The strategy used by the L4.Verified project was to translate the C implementation of the μ-kernel into the Simpl language [20], which in turn was implemented using High-Order Logic (HOL) on the Isabelle theorem prover [19]. Then, invariants about the representation of the operating system components were proved with Isabelle. Finally, the Simpl description was demonstrated to refine the kernel's specification described in HOL. The approach has the advantage of keeping the specification separated from the implementation and described at a higher level of abstraction, but requires several refinements that must be carried out manually and that are out of reach for most operating system developers.

Cohen has verified the functional correctness of Microsoft's Hyper-V virtualization platform and also of SYSGO's embedded real-time operating system PikeOS [7]. His strategy was based on a logic to express Locally Checked Invariants (LCI) that can be used to assign predicates to components. Predicates define a sort of contract that can be subsequently verified by the VCC verification environment [6]. The approach shares some aspects with the one proposed here, but each function must be verified in isolation, using only the contracts of called functions and invariants of types used in its code.

Gotstman and others propose the modular verification of preemptive kernels running in multiprocessor machines [13]. Gotsman's approach propose two proof systems: a high-level one, in which all processes have their own virtual CPU, and a low-level, in which the number of processors is fixed. The high-level proof system is used to reason about processes while the lower-level proof targets the scheduler itself. This separation and the isolation of the scheduler from the rest of the kernel allows for modular verification. Although the proposal is to verify mainstreams operating systems such as Linux and FreeBSD, all theory presented in the paper is developed around an assembly language for a fictional machine.

Differently from these related works, our proposal aims at verifying functional correctness and safety of individual operating system components described at system-level. Additionally, we address verification without requiring OS developers to specify the behavior of components using specific languages and tools. In our approach, which is detailed in the next section, contracts are written in the same language as components and no refinements have to be proven.

3. VERIFICATION APPROACH

The convergence between software and hardware in the design of embedded systems calls for embedded operating systems whose parts can also be freely shifted from one domain into the other. Accordingly, the formal verification of such systems must now be carried out independently of the instantiation domain. In this scenario, the implementation itself must be domain-independent. Therefore, we propose a strategy to handle these cases that consists of three main steps: (i) writing contracts for the components that will be verified, (ii) instrumenting components with such contracts, (iii) performing software bounded model checking to verify if components respect their contracts.

The fact that modern ESL design tools such as CatapultC [2] are able to synthesize hardware directly from C++ descriptions enables us to propose a single-language strategy. Components and contracts are both written in C++ on a seamless way inspired by the Eiffel programming language [17]. We propose contracts to be 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 pre and postconditions. Class invariants are defined as a set of assertions that are always true for all instances of that class. The preconditions of a method define a set of assertions that must be satisfied before it is executed. Conversely, the postconditions of a method define a set of assertions that must be true in case the method invocation terminates normally (i.e. without triggering exceptions, which have not yet been addressed in our approach). Pre and postcondition assertions can reason about the method's parameters, return value, and object state. Figure 1 shows an example of contract for the method `insert` of a `Queue` class. The `invariants` method defines the class' invariants, which, in the example, state that a queue of this class can never be empty.

Writing contracts directly on the component implementa-

```
void Queue::insert (T 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));
}

void invariants ()
{
  assert ( size () >= 1);
}
```

Figure 1: Example of contract for the `insert()` method of a `Queue` class.

tion has the advantage of keeping specification and implementation close to each other, always at hand to be informally verified by programmers during development. However, writing assertions directly on the component code can impose additional run-time overhead to its instances. The code added to components to specify their contracts can be partially eliminated by the macro mechanism traditionally implemented in the standard C library. Since contracts are only used during the verification phase, the `libc`'s `assert` macro (declared in `assert.h`) can be defined to be void for the compilation of the final system. Nevertheless, contracts that require the declaration of new variables cannot be handled in this way. For example, the expression

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

in the postcondition of method `insert` in Figure 1 depends on the declaration of the (*size_at_pre*) variable. This variable was not present in the original method and was only included to support the method's contract. In such cases, Aspect-oriented Programming (AOP) techniques can be of great help. Both assertions and associated variables can be injected into the component's code specifically for the verification stage and left out for the final system instantiation.

In our approach, we resort to the *Scenario Adapter* pattern [10], proposed by Fröhlich, to realize a *Verified Scenario* for specific components. As it can be seen from Figure 2, the client (*Client*) of a component accesses its methods though the scenario adapter (`Scenario_Adapter`), which encapsulates the component in a scenario defined by a collection of aspect programs meant to properly wrap each component method invocation, thus imputing them the semantics established by that scenario. The `Scenario_Adapter` applies the pattern shown at the implementation of the method `operation()` for each method of the component. It calls the preconditions of such method before calling the actual method and then calls postconditions after the method is called. The actual method is called through delegation (`IMP-`

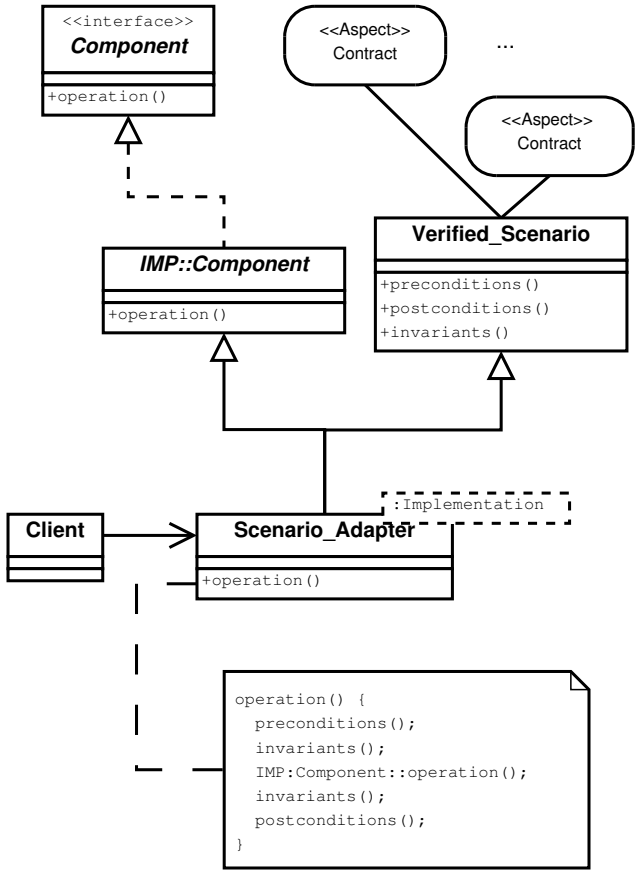

Figure 2: The Verified Scenario to make components verifiable using AOP techniques.

`::Component::operation()`). It also calls the `invariants()` just before and just after calling the actual method to ensure that the method does not violate the class invariants. The `Verified_Scenario`, which can be customized for each individual component, combines all invariants, pre and postconditions through inheritance to define a deployment scenario in which that component can be verified. Assertions touching cross-cutting properties or referring recurring properties can be freely reused in several `Verified_Scenario` wrappers[1].

The main advantage of using the scenario adapter pattern to instrument a component with its contract is the possibility of easily enabling and disabling the instrumentation as needed. In order to enable the instrumentation, one can use a type alias as the following:

```
namespace IMP {
    #include <component>
}
typedef Scenario_Adapter<IMP::Component> Component;
```

As the scenario adapter implements the public interface of the *Component* interface, the client class accesses the verified version of the component just as it accesses the original

[1]Actually, a single implementation of `Verified_Scenario` can be defined for all components. It is subsequently specialized by using a metaprogrammed list of conditions and invariants that must be applied to each method of each component.

(and final) component. Once the component has been instrumented with its contract, that is, once it is put in its `Verified_Scenario`, verification can be carried out by traditional software model checkers.

Besides instrumenting the component implementation with its contract, AOP techniques can also be deployed to isolate the component target of verification from other components. Isolation is essential to enable a modular verification of components, in which not-yet-verified components can be disconnected from the component currently being verified. Furthermore, verifying a whole embedded operating system at once is a defying process. Even the L4.Verified project, which verified a whole μ-kernel, used isolation to some extent as they pushed I/O and interrupt handling to well-defined and well-behaved places inside the kernel [14]. To make the case for modular verification, consider resource scheduler depicted in Figure 3. The `Scheduler` class is a parametric class designed and implemented to schedule all sort of resources. However, its traditional deployment as a `Thread` scheduler brings about several complex issues, including context exchange, interrupt handling and many other asynchronous events that so often challenge formal method researchers.

In order to isolate the `Scheduler` from the `Thread` implementation (and from any other complex resource being scheduled), the `Verified_Scenario` forces it to be instantiated with a `Dummy_Thread` class. The `Dummy_Thread` class implements the same interface of class `Thread`, represented in Figure 3 by the association class `Scheduling_Criteria`. In this design, all schedulable resources must define the `int()` operator, thus enabling them to be properly ordered by the scheduler in accordance with the scheduling police in force. Scheduling policies are specified by controlling the mapping of objects representing resources into the totally ordered set of integer numbers $\mathbb{Z}$. Since any interaction between the scheduler and the objects being scheduled is performed through this narrow interface (i.e. `operator int()`), replacing `Thread` by `Dummy_Thread` does not impair the verification of `Scheduler` as long as the total ordering property holds, something ensured by the compiler from the definition of type `integer`.

It is important to notice that the isolation of components described here to illustrate the proposed approach for verification can only be done efficiently for components designed to be reused in a variety of scenarios. The scheduler discussed here was designed following the ADESD methodology [18] with that purpose in mind. It is unlikely that the thread scheduler of a monolithic (i.e. a non-component-based) μ-kernel could be isolated in this manner. Monolithic designs are more likely to be verified in an all-or-nothing approach or perhaps to be refactored as a set of semi-independent modules. In any case, isolation for verification can never be taken independently from component design since the effect of isolation over each individual property being verified must be taken into account.

We have so far demonstrated our approach with the C Bounded Model Checker (CBMC) [15]. The component implementation instrumented with its contract is translated by CBMC into a logical formula through a process called *symbolic simulation*[5]. While performing this translation, CBMC also performs static analysis of the source code generating properties to be checked. Among these properties, one can find array bound and pointer safety plus user defined assertions, which in our case correspond to the contracts of

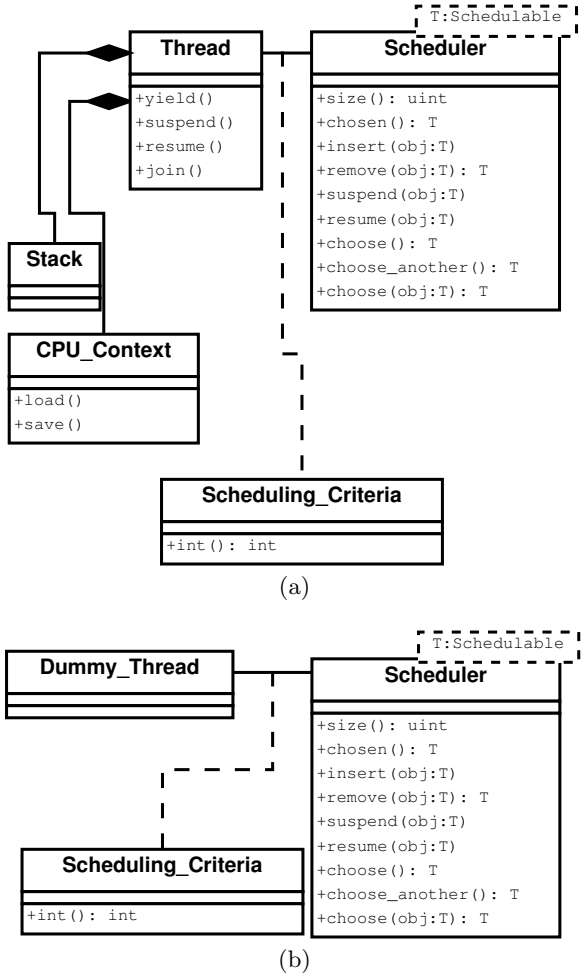

Figure 3: Component isolation for verification. (a) Before isolation (b) After isolation.

components. Subsequently, the logical formula representing the component is combined with the properties generated by CBMC to yield verification conditions. Finally, CBMC invokes a SAT solver to determine if all the properties defined for the component being verified hold. In case a property does not hold, a counter-example is generated showing the execution path where the property became false. If all properties are true, the verification ends successfully proving that the component fulfills both its contract and the automatically generated safety properties.

The next section describes the verification of the scheduler introduced in this section in details aiming at illustrating and corroborating the approach introduced here.

4. CASE STUDY

We have chosen the scheduler of the Embedded Parallel Operating System (EPOS) [9] as a case study to evaluate the verification strategy proposed here because it has been extensively investigated as a system-level design artifact in the realm of embedded systems [16]. It was first implemented as a software/hardware hybrid component following a design that allowed it to be described either in C++ or in VHDL. Subsequently, its C++ description has been demonstrated

to be directly synthesizable using ESL tools [8]. Moreover, the scheduler is a key component of an operating system, handling queues and dealing with polices under severe performance constraints.

An overview of EPOS scheduler is shown in Figure 4. The scheduler itself is designed to be independent from the resources (i.e. objects) it schedules—threads in this case. As it was discussed in the previous section, queue management and scheduling policies are also kept independent of each other as much as possible. They are connected only by the `int()` operator defined in `Scheduling_Criteria`, which maps schedulable objects into the totally ordered set of integer numbers $\mathbb{Z}$, thus defining a scheduling policy algorithm. Total ordering dictates that higher ranked objects have precedence over lower ranked ones in accordance with the ordering in $\mathbb{Z}$. It does not require schedulable objects to be uniquely ranked. Ties are either handled on a round-robin manner (i.e. queue reinsertion at tail) or first-in-first-out (i.e. queue reinsertion at head) according with a user visible configurable feature of the scheduler.

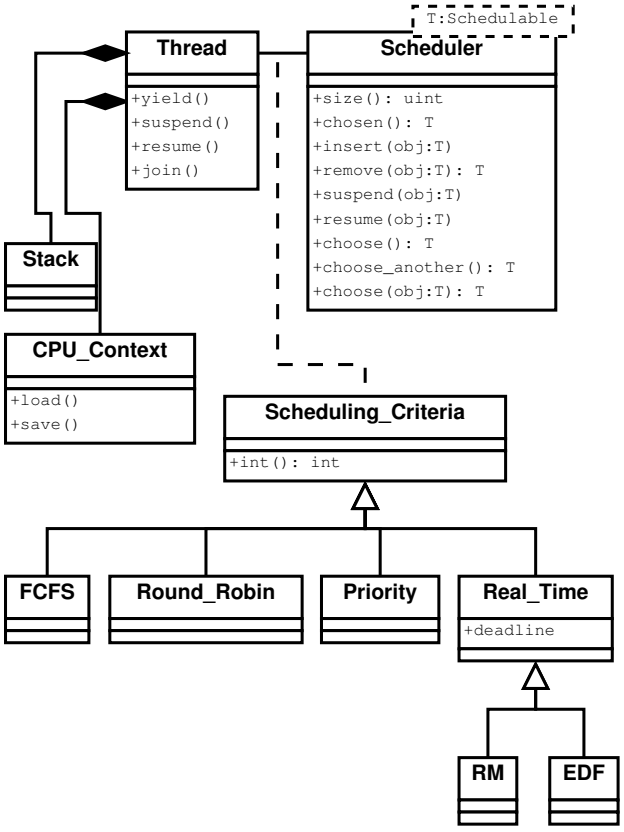

Figure 4: Overview of EPOS scheduler design.

To verify the functional correctness and safety of EPOS scheduler, we must first specify its contract. A single class invariant was defined. It dictates that one cannot remove more schedulable objects from the scheduler than have been inserted into it.

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

Afterwards, we defined C++ assertions representing pre and postconditions for each public method of class `Scheduler`. They are shown at the Table 1. Variables prefixed `at_pre` hold the value of the corresponding property obtained at the precondition, that is, before the method execution. The private method `contains()` checks whether the object given as argument is contained in the scheduler. The safety properties of the scheduler contract related to array bounds and pointer safety are automatically generated by CBMC and are not shown in the table. The specified contract was incorporated into EPOS scheduler implementation using the *verified scenario* described in Section 3. No modifications of any kind were made in the scheduler itself, thus ensuring that the verified version is equivalent to the final version (of which assertions are striped).

After being instrumented with its contract, EPOS scheduler was submitted to CBMC for formal verification. CBMC generates VCCs pertaining the functional correctness properties specified by the contract and also VCCs related to array bound and pointer safety. In our case, 2344 VCCs were generated. After generating VCCs, CBMC performs a simplification phase based on constant propagation and expression rewriting. After that phase, 2121 VCCs remained. The whole verification process took around 10 minutes running on an Intel Core 2 at 2.83GHz. At the end of the process, CBMC indicated that all VCCs held, confirming that the scheduler respected its contract and also the no safety properties are violated. The verified code was subsequently compiled using GCC yielding a software instance and using the CatapultC HLS tool [2] yielding a hardware instance of the scheduler.

5. CONCLUSION

In this paper, we have introduced an approach to formally verify functional correctness and safety properties of embedded operating system components described at System-Level. The functional correctness properties of a component are expressed by a contract containing assertions that represent class invariants, pre, and postconditions for each method declared in the component's interface. The strategy of enriching components with contracts was inspired by the Eiffel programming language designed by Meyer. The safety properties are meant to be automatically generated by a model checker. Components target of verification are instrumented with their contracts using the *Verified Scenario* pattern, which allows for assertion reuse and component isolation for modular verification. Components instrumented with contracts as described here can be submitted to the CBMC model checker for formal verification. Subsequently, the *Verified Scenario* can be disabled to eliminate any overhead caused by the instrumentation, thus yielding a production version of the component with absolutely the same properties.

We have chosen the C++ programing language for implementing components and contracts because it is a strong contender in the System-Level realm. Components described in C++ following specific design methodologies can now be automatically synthesized both as software, hardware or arbitrary combinations of both. Nevertheless, we believe that the principles behind our proposal can easily be adapted to other languages and model checkers. The verified scenario construct can be directly applied to SystemC, since it is implemented as a C++ library. For SpecC, the verified scenario can be applied by using aggregation instead of inheritance in order to collect the verification aspect programs used by the component's verified scenario, and sub-

Table 1: Pre and postconditions to be verified for EPOS scheduler's methods.

Method	Preconditions	Postconditions
unsigned int size() const	*none*	*none*

T * volatile chosen() const	*none*	if(!empty())
		  assert(contains(result);
		else
		  assert(result == null);

void insert(T * obj)	**assert**(obj != null); **assert**(!contains(obj));	**assert**(size() == size_at_pre + 1); **assert**(contains(obj));

T * remove(T * obj)	**assert**(obj != null);	**assert**(!contains(obj));
		if(at_pre.contains(obj))
		  assert(size() == size_at_pre - 1)
		else
		  assert(result == null);

void suspend(T * obj)	**assert**(obj != null);	**assert**(!contains(obj)); if(at_pre.contains(obj)) **assert**(size() == size_at_pre - 1)
void resume(T * obj)	*same as insert*	*same as insert*

T * choose()	*none*	**assert**(size() == size_at_pre)
		if(empty())
		  assert(result == null)
		else
		  assert(contains(result))

T * choose_another()	*same as choose*	*same as choose*

T * choose(T* obj)	**assert**(obj != null);	**assert**(size() == size_at_pre);
		if(at_pre.contains(obj)) {
		  assert(contains(result));
		  assert(result==obj);
		} else
		  assert(result == null);

sequently to combine it with the *Scenario Adapter*. Then, any model checker supporting such languages and supporting user-defined assertions can be used to proceed a formal verification.

The isolation of components for modular verification, however, is something that is made possible by design and not simply by deploying tools or language patterns. The scheduler verified in this article was designed following the *Application-Driven Embedded Systems Design* (ADESD) methodology [18] to be independent from the envisioned deployment scenarios. Scenario independence is attained by minimizing the connections between components and by the encapsulation of scenario dependencies in aspect programs. Such design principles produced a scheduler with a lean interface and with whose external dependencies were restricted to a single method (`operator int()`. It is unlikely that the thread scheduler of a monolithic (i.e. a non-component-based) μ-kernel could be isolated in this manner. Monolithic designs are more likely to be verified in an all-or-nothing approach and probably would not benefit much from the approach proposed here.

6. REFERENCES

[1] N. Blanc, D. Kroening, and N. Sharygina. Scoot: a tool for the analysis of systemc models. In *Proc. of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, pages 467–470, 2008.

[2] Calypto Design Systems. CatapultC Synthesis, 2011. http://www.calypto.com/.

[3] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. Kratos: a software model checker for systemc. In *Proc. of the 23rd international conference on Computer aided verification*, pages 310–316, 2011.

[4] E. Clarke, H. Jain, and D. Kroening. Verification of specc using predicate abstraction. *Form. Methods Syst. Des.*, 30(1):5–28, Feb. 2007.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.

[6] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. Vcc: A practical system for verifying concurrent c. In *Proc. of the 22nd International Conference on Theorem Proving in Higher Order Logics*, pages 23–42, 2009.

[7] E. Cohen, M. Moskal, W. Schulte, and S. Tobies. Local verification of global invariants in concurrent programs. In *Proc. of the 22nd international conference on Computer Aided Verification*, pages 480–494, 2010.

[8] J. P. P. Flor, T. R. Mück, and A. A. Fröhlich. High-level Design and Synthesis of a Resource Scheduler. In *18th IEEE International Conference on*

Electronics, Circuits, and Systems, pages 736–739, Beirut, Lebanon, Dec. 2011.

[9] A. A. Fröhlich. *Application-Oriented Operating Systems*. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[10] A. A. Fröhlich and W. Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, July 2000.

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

[12] 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*, 28(10):1517–1530, Oct. 2009.

[13] A. Gotsman and H. Yang. Modular verification of preemptive os kernels. *SIGPLAN Notices*, 46(9):404–417, Sept. 2011.

[14] 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 *Proc. of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, New York, NY, USA, 2009.

[15] D. Kröning. The cbmc homepage, 2012.

[16] 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*, pages 382–387, Vancouver, Canada, Aug. 2009.

[17] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[18] T. R. Mück, M. Gernoth, W. Schröder-Preikschat, and A. A. Fröhlich. Implementing OS Components in Hardware using AOP. *SIGOPS Operating Systems Review*, 46(1):64–72, 2012.

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

[20] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.