Automating Formal Modular Verification of Asynchronous Real-Time Embedded Systems *

Pao-Ann Hsiung and Shu-Yu Cheng
Department of Computer Science and Information Engineering
National Chung Cheng University, Chiayi–621, Taiwan, ROC
E-mail: hpa@computer.org

Abstract

Most verification tools and methodologies such as model checking, equivalence checking, hardware verification, software verification, and hardware-software coverification often flatten out the behavior of a target system before verification. Inherent modularities*, either explicit or implicit, functional or structural, are not exploited by these tools and algorithms. In this work, we show how* assume-guarantee reasoning *(AGR) can be used for such exploitations by integrating AGR into a verification tool. Targeting at real-time embedded systems, we propose procedures to* automatically *generate assumptions, guarantees, and time constraints, which otherwise require manual efforts and human creativity. Through a complex but comprehensible real-time embedded system example such as a* Vehicle Parking Management System *(VPMS), we illustrate the feasibility of the AGR approach and the extremely large reduction possible in state-space sizes when AGR is applied. Due to AGR, we also found five errors in the VPMS design using much* lesser *CPU time and memory space than possible without AGR.*

1 Introduction

The theory behind *Assume-Guarantee Reasoning* (AGR) has been well-studied and can be traced back to Misra and Chandy's *assumption-commitment* approach [24] and Jones' *rely-guarantee* approach [19] proposed around two decades ago. Though AGR has a long history, yet it has been "more widely studied than actually used" [26]. Theoretically, AGR states that a system can be verified by first decomposing it into constituent parts, second the parts are individually verified such that each part satisfies a guarantee G only if its environment satisfies an assumption A, and finally discharging all the assumptions made for each component using a *circular induction over time*. This reasoning will be explained in more details in Section 3. The main benefit of this approach is that the explicit construction of the system global state-space, which is usually of an exponentially large size, can be avoided [16, 17]. Thus, verification scalability is increased through the application of AGR.

Only in the recent few years has there been some applications of the AGR technique to real-world systems such as asynchronous systems [1, 2], synchronous reactive systems [7, 8, 18], Tomasulo's algorithm [22], a pipelined implementation of a directory-based coherence protocol in Silicon Graphics Origin 2000 servers [10], a VGI dataflow processor array designed by the Infopad project at U. C. Berkeley [11], pipelined implementation of an ISA architecture [14], audio output interface of a multimedia extension SoC [25], and a software supervisor for a multi-user phone system [28].

The AGR technique has also been extended in several ways, for example, to accomodate multiple constraints on a single output port [22], branching time refinement [15], different implementation and specification time scales [13], and liveness constraints [23].

The application of AGR can be semi-automatically performed by a user of the MOCHA tool [9, 3] through its proof manager, but the user is still burdened with the task of constructing *abstraction* and *witness* modules [12], which in general requires human creativity. Recently, there are some works on mechanizing the construction of both abstraction modules [4] and witness modules [6]. Automation for the application of AGR has been greatly enhanced by such mechanizations. Nevertheless, the automation is still limited to refinement checking.

All the above-cited previous works show that the AGR technique is gaining importance due to the increase in system complexity. Nevertheless, the above literatures mainly consists of case studies, where it is

*This work was supported in part by project grant NSC 91-2213-E-194-008 from the National Science Council, Taiwan, ROC.

shown how AGR can be applied to a particular system. As detailed above, the application of AGR is also limited to refinement checking in the current version of the MOCHA tool. In our present work, firstly, we show how the application of AGR can be generalized for the verification of a typical real-time embedded system. We propose automating the application of AGR not only in *refinement checking*, but also in *invariant checking*. Secondly, we show how assumptions, guarantees, and time constraints can be automatically generated for a real-time embedded system. Finally, we illustrate through an example how AGR helps in uncovering design faults using lesser CPU time and memory space.

This article is organized as follows. Section 2 will formulate the problem to be solved and describe the system model along with an example of a real-time embedded system. Section 3 will illustrate how assume-guarantee reasoning can be applied to the formal verification of SoC, along with the automatic generation of assumptions and guarantees. Section 4 will give the verification results conducted for the VPMS example. Section 5 will conclude the article with some research directions for future work.

2 System Model

Our target system for verification is a *Real-Time Embedded System* (RTES), which we basically view as a collection of embedded hardware components, software components, and interfaces. Our real-time embedded system model is based on the timed automata model [5], which is defined as follows, where the sets of integers and non-negative real numbers are denoted by $\mathcal{N}$ and $\mathcal{R}_{\geq 0}$, respectively.

Definition 1 *:* **Mode Predicate**
Given a set C of clock variables and a set D of discrete variables, the syntax of a *mode predicate* η over C and D is defined as: $\eta := \mathit{false} \mid x \sim c \mid x - y \sim c \mid d \sim c \mid \eta_1 \wedge \eta_2 \mid \neg\eta_1$, where $x, y \in C$, $\sim \in \{\leq, <, =, \geq, >\}$, $c \in \mathcal{N}$, $d \in D$, and η_1, η_2 are mode predicates. ‖

Let $B(C, D)$ be the set of all mode predicates over C and D.

Definition 2 *:* **Timed Automaton**
A *Timed Automaton* (TA) is a tuple $\mathcal{A}_i = (M_i, m_i^0, C_i, D_i, L_i, \chi_i, E_i, \lambda_i, \tau_i, \rho_i)$ such that: M_i is a finite set of modes, $m_i^0 \in M$ is the initial mode, C_i is a set of clock variables, D_i is a set of discrete variables, L_i is a set of synchronization labels, $\chi_i : M_i \mapsto B(C_i, D_i)$ is an *invariance* function that labels each mode with a condition true in that mode, $E_i \subseteq M_i \times M_i$ is a set of transitions, $\lambda_i : E_i \mapsto L_i$ associates a synchronization label with a transition, $\tau_i : E_i \mapsto B(C_i, D_i)$ defines the transition triggering conditions, and $\rho_i : E_i \mapsto 2^{C_i \cup (D_i \times \mathcal{N})}$ is an *assignment* function that maps each transition to a set of assignments such as resetting some clock variables and setting some discrete variables to specific integer values. ‖

Using the above TA definition, our system model can be defined as follows.

Definition 3 *:* **Real-Time Embedded System (RTES)**
A *Real-Time Embedded System* is defined as a collection of hardware, software, and interface components. Each component is modeled by one or more timed automata. A system is modeled by a network of communicating timed automata. Notationally, if a system $\mathcal{S}$ has a set of hardware components $\{H_1, H_2, \ldots, H_n\}$ and a set of software components $\{S_1, S_2, \ldots, S_m\}$, then $\mathcal{S} = H_1 \| H_2 \| \ldots \| H_n \| S_1 \| S_2 \| \ldots \| S_m$, where $\|$ is a parallel composition operator resulting in the concurrent behavior of its two operands. If H_i is modeled by a TA A_{H_i}, $1 \leq i \leq n$, and S_j is modeled by a TA A_{S_j}, $1 \leq j \leq m$, then the TA defined by $A_S = A_{H_1} \times \ldots \times A_{H_n} \times A_{S_1} \times \ldots \times A_{S_m}$ is a TA model for system $\mathcal{S}$, where $\times$ is the Cartesian product operator for two timed automata. Concurrency semantics is defined as follows. Two concurrent transitions with the same synchronization label are represented by a single synchronized transition. Two concurrent transitions without any synchronization label are represented by interleaving them, resulting in possibly two different paths (computations). ‖

For simplicity, it is assumed that a single hardware or software component is modeled by a single TA, instead of the more general case of one or more TA. The above definition can be easily extended to the general case.

An embedded real-time system called *Vehicle Parking Management System* (VPMS) [20, 21] will be used to illustrate our verification methodology throughout this article.

VPMS controls the entry and exit of vehicles into and from a parking garage or lot. Functionally, it consists of the three subsystems: an ENTRY Management Subsystem, which controls the entry of vehicles into a garage such that each driver gets a parking ticket with an entry time stamp, an EXIT Management Subsystem, which controls the exit of vehicles from a garage such that only drivers with a valid paid ticket gets permission to exit, and a DISPLAY Subsystem, which indicates the number of vacant parking spaces currently available in a garage or lot.

The architecture of VPMS is illustrated in Figure 1

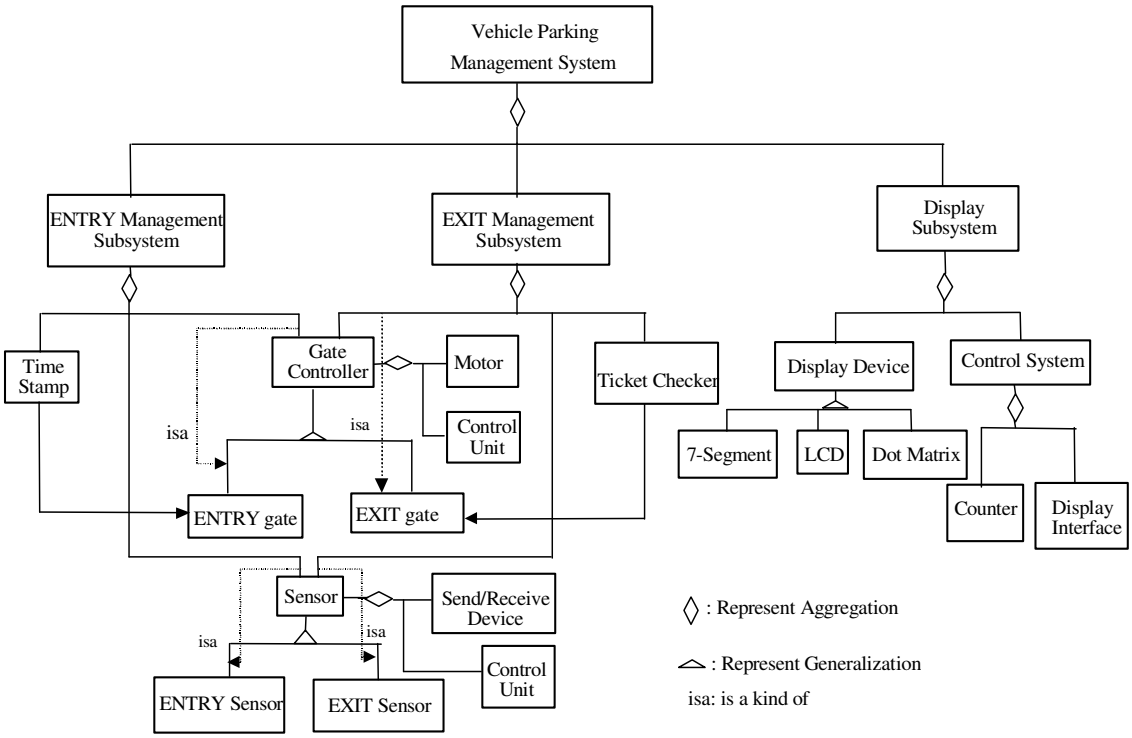

Figure 1: Vehicle Parking Management System

using the *Unified Modeling Language* (UML). An ENTRY (or an EXIT) subsystem consists of three parts: a ticket processor, a motor-controlled gate, and a set of sensors. Constraints for the VPMS system include: a maximum system cost of $1,300, a maximum ticket emission time of 20 ns, a maximum display response time of 250 ns. VPMS is modeled using five TA: one for each of the three subsystems, namely ENTRY, EXIT, and DISPLAY, and two for the environment, which includes the user and other external devices such as the Display Board. Further details on VPMS can be found in [20, 21].

3 Assume-Guarantee Verification

Assume-guarantee reasoning (AGR) is the dual counterpart to formal verification just as *divide-and-conquer* is to discrete optimization. Informally, AGR combines verification results of each constituent part of a system to make conclusions on the verification of the whole system, instead of directly verifying the full system. AGR can be beneficial in terms of higher verification scalability, provided the size of the state-space for the individual verification of each constituent part is significantly smaller than that for the full system. Furthermore, the adoption or application of AGR is often restrained by the necessity for human creativity in the following tasks: (1) In *refinement checking*, *abstraction modules* [4] and *witness modules* have to be constructed [6], and (2) In *invariant checking*, assumptions and guarantees have to be generated.

The rules for assume-guarantee reasoning appear in several different forms in the literature. Here, we give the form of rules on which our work is based. As shown in Equation (1), we have extended the rules for applying AGR to invariant checking from [28] by including timing constraints. A system $\mathcal{S}$ has an assumption A, a guarantee G, and a Boolean timing constraint T. Each component of the system also has an assumption A_i, a guarantee G_i, and a timing constraint T_i. From Equation (1), we see there are $2n + 1$ premises to be satisfied for a system with n components to be completely verified and to arrive at the conclusion $A \rightarrow_T G$, where $\rightarrow_T$ denotes logical implication while satisfying time constraints T. The first set of n premises $A_i \rightarrow_{T_i} G_i$ gives the rule for verifying that each component satisfies its own guarantee G_i under its assumption A_i and time constraint T_i. The second set of n premises constitute the discharging of all the assumptions by ensuring that each A_i can be implied by the system assumption A and the guaran-

tees $G_j, j \neq i$ of the other components under the time constraints T and $T_j, \forall j \neq i$. The last premise simply states that the system behavior G is constructed from a conjunction of the behaviors of each system component G_j. This last premise must be considered and ensured while G and each G_j are being constructed.

$$\frac{\begin{array}{rll} A_i & \longrightarrow_{T_i} & G_i, \\ A \wedge \bigwedge_{j \neq i} G_j & \longrightarrow_{T \wedge \bigwedge_{j \in \{1,\ldots,n\}} T_j} & A_i, \\ \bigwedge_{j \in \{1,\ldots,n\}} G_j & \longrightarrow_{T \wedge \bigwedge_{j \in \{1,\ldots,n\}} T_j} & G \end{array}}{\begin{array}{rll} A & \longrightarrow_{T} & G \end{array}} \quad (1)$$

To apply and take advantage of assume-guarantee reasoning in verifying a complex system, *assumptions*, *guarantees*, and *time constraints* are required for each system component and for the system environment. Our algorithm to automatically generate them for a system component is as detailed in Table 1.

4 Verification Results for the VPMS Example

We applied the assume-guarantee reasoning principles to the *Vehicle Parking Management System* (VPMS) [20, 21] example, which was introduced in Section 2. After applying the algorithm from Table 1, the assumptions, guarantees, and time constraints for VPMS were generated, part of which are given in Table 2. There are three computation runs for each of ENTRY and EXIT subsystems, and four computations runs for the DISPLAY subsystem. As given in the last two rows of Table 2, namely Entry_Environment and Exit_Environment, the assumptions, guarantees, and time constraints for the system environment were derived from user-given requirements (see Section 2).

The AGR rules for invariant checking given in Equation (1) were all checked with the assumptions, guarantees, and time constraints of VPMS (Table 2). There were five errors found as follows.

- *Component Assumption Error*: While using the second rule (Equation (1)) for discharging the component assumption with time-constraints in the Entry component (see first row of Table 2), two errors were found in the Entry assumptions count_above_zero? and count_zero? with time constraints δ(count_request!, count_above_zero?) $= [200, 200]$ and δ(count_request!, count_zero?) $= [200, 200]$. It was found that these time constraints could not be satisfied because of contradiction with the component guarantees count_above_zero! and count_zero! with time constraints δ(count_request?, count_above_zero!) $= [18, 18]$ and δ(count_request?, count_zero!) $= [18, 18]$ in the Display component. Solutions to the first error could consist of changing either of the two time constraints, but because the signal count_above_zero could be output (guaranteed) by 18 ns, our solution to this error was to change the time constraint of the Entry component to δ(count_request!, count_above_zero?) $= [0, 200]$. Similarly, our solution to the second error was to change the time constraint of the Entry component to δ(count_request!, count_zero?) $= [0, 200]$.

- *Environment Guarantee Errors*: While using the third rule (Equation (1)) for checking whether the environment guarantee was conjunctively implied by the component guarantees, two errors were found in the environment guarantees ent_update_dboard? and ex_update_dboard? with time constraints δ(take_ticket!, ent_update_dboard?) $= [0, 250]$ and δ(ticket_ok?, ex_update_dboard?) $= [0, 250]$. The time constraints were originally derived from the user-given constraint that the maximum display response time should be 250 ns. These time constraints could not be satisfied by the system components. For example, consider the first time constraint mentioned above. The conjunction of δ(take_ticket?, car_in!) $= [244, \infty)$ from Entry with δ(car_in?, ent_update_dboard!) $= [42, 142]$ from Display results in δ(take_ticket!, ent_update_dboard?) $= [286, \infty)$, which does not satisfy the user-given constraint of 250 ns maximum. Solutions to this error could consist of changing either component or environment time constraints, but because the component constraints could not be changed due to physical device restrictions, our solution was to ask the user to relax his/her constraint to at least 286 ns.

- *Environment Assumption Error*: While using the second rule (Equation (1)) for discharging the basic assumption push_button? with time constraint δ(car_in!, push_button?) $= [244, \infty)$ in the Entry_1 schedule of the Entry component (see first row of Table 2), it was found that the time constraint could not be guaranteed unless there was an environment assumption δ(ent_update_dboard?, push_button!) $= [202, \infty)$. This is because there is only a time constraint between signals car_in and ent_update_dboard (i.e.,

Table 1: Automatic Generation of Assumptions, Guarantees, and Timing Constraints

Gen_Comp_AG(X_i)
$X_i \in \mathcal{S} = \{H_1, \ldots, H_n, S_1, \ldots, S_m\}$;
{
$A_i = \{\}$; $G_i = \{\}$; $T_i = \{\}$; (1)
$schedule\_set$ = All_Finite_Schedules(X_i, m_i^0); (2)
while (ψ = One_Finite_Schedule$(schedule_set) \neq$ NULL) { (3)
$last\_signal$ = NULL; $first\_time = second\_time$ = NULL; (4)
// start generating assumption and guarantee (5)
while (γ = Get_Signal$(\psi) \neq$ NULL) { (6)
if ($last\_signal ==$ NULL and type$(\gamma) ==$ out) (7)
return Unsupported_System_ERROR; // schedule begins with output signals (8)
switch (type(γ)) { (9)
case 'in': (10)
if ($last\_signal ==$ in) $Basic\_a = Basic\_a \oplus \gamma$; // $\oplus \in \{\prec, \preceq\}$ (11)
else { (12)
if ($Basic\_a \neq$ NULL) $Schedule\_a = \langle Schedule\_a, Basic\_a \rangle$; (13)
$Basic\_a = \gamma$; (14)
$last\_signal$ = in; } break; (15)
case 'out': (16)
if ($last\_signal ==$ in) { (17)
if ($Basic\_g \neq$ NULL) $Schedule\_g = \langle Schedule\_g, Basic\_g \rangle$; (18)
$Basic\_g = \gamma$; (19)
$last\_signal$ = out; } (20)
else $Basic\_g = Basic\_g \oplus \gamma$; break; } } (21)
if ($Basic\_a \neq$ NULL) $Schedule\_a = \langle Schedule\_a, Basic\_a \rangle$; (22)
if ($Basic\_g \neq$ NULL) $Schedule\_g = \langle Schedule\_g, Basic\_g \rangle$; (23)
if $|Schedule\_a| \neq |Schedule\_g|$ return Unsupported_System_ERROR; (24)
else { $A_i = A_i \cup Schedule\_a$; $G_i = G_i \cup Schedule\_g$; } (25)
// start generating time-constraints (26)
while (ζ = Get_Temporal_Signal$(\psi) \neq$ NULL) { (27)
if (there is signal with temporal value in ζ) (28)
$first\_time = \zeta$; (29)
if ($first\_time \neq$ NULL and there is signal with temporal value in ζ) (30)
$second\_time = \zeta$; (31)
if ($first\_time \neq$ NULL and $second\_time \neq$ NULL){ (32)
$Basic\_t$ = Evaluate_Time_Constraint$(first_time, second_time)$; (33)
$Schedule\_t = Schedule\_t \wedge Basic\_t$; (34)
$first\_time = second\_time$ = NULL; } } (35)
$T_i = T_i \cup Schedule\_t$; } (36)
return (A_i, G_i, T_i); (37)
}

Table 2: Some Assumptions, Guarantees, and Time Constraints for VPMS

Subsystem	Schedule #	**Assumption**(A), **Guarantee** (G), **Time Constraints*** (T)
Entry	Entry_1	$A:$ $\langle$push_button?, count_above_zero?, take_ticket?$\rangle$ $G:$ $\langle$count_request!, ticket_out!, car_in!$\rangle$ $T:$ δ(count_request!, count_above_zero?) $= [200, 200] \wedge$ δ(take_ticket?, car_in!) $= [244, \infty) \wedge$ δ(car_in!, push_button?) $= [244, \infty)$
	Entry_2	$A:$ $\langle$push_button?, count_zero?$\rangle$ $G:$ $\langle$count_request!, no_ticket_out!$\rangle$ $T:$ δ(count_request!, count_zero?) $= [200, 200]$
	Entry_3	$A:$ $\langle$push_button?, count_above_zero?, take_ticket?$\rangle$ $G:$ $\langle$count_request!, ticket_out!, ent_time_out!$\rangle$ $T:$ δ(count_request!, count_above_zero?) $= [200, 200] \wedge$ δ(ent_time_out!, push_button?) $= [244, \infty)$
Exit	Exit_1	$A:$ $\langle$ticket_insert?$\rangle$, $G:\langle$ticket_ok! $\prec$ car_out!$\rangle$ $T:$ δ(ticket_ok!, car_out!) $= [244, \infty) \wedge$ δ(car_out!, ticket_insert?) $= [244, \infty)$
	Exit_2	$A:\langle$ticket_insert?$\rangle$, $G:\langle$ticket_error!$\rangle$
	Exit_3	$A:$ $\langle$ticket_insert?$\rangle$, $G:\langle$ticket_ok! $\preceq$ ex_time_out!$\rangle$ $T:$ δ(ex_time_out!, ticket_insert?) $= [244, \infty)$
Display	Display_1	$A:$ $\langle$initialize?, car_in?$\rangle$ $G:$ $\langle$reset_dboard!, ent_update_dboard!$\rangle$ $T:$ δ(initialize?, reset_dboard!) $= [0, 100] \wedge$ δ(car_in?, ent_update_dboard!) $= [42, 142]$
	Display_2	$A:$ $\langle$initialize?, count_request?$\rangle$ $G:$ $\langle$reset_dboard!, count_zero!$\rangle$ $T:$ δ(initialize?, reset_dboard!) $= [0, 100] \wedge$ δ(count_request?, count_zero!) $= [18, 18]$
	Display_3	$A:$ $\langle$initialize?, count_request?$\rangle$ $G:$ $\langle$reset_dboard!, count_above_zero!$\rangle$ $T:$ δ(initialize?, reset_dboard!) $= [0, 100] \wedge$ δ(count_request?, count_above_zero!) $= [18, 18]$
	Display_4	$A:$ $\langle$initialize?, car_out?$\rangle$ $G:$ $\langle$reset_dboard!, ex_update_dboard!$\rangle$ $T:$ δ(initialize?, reset_dboard!) $= [0, 100] \wedge$ δ(car_out?, ex_update_dboard!) $= [42, 142]$
Entry_Environment	Entry_Env_1	$A:$ $\langle$push_button!, take_ticket!$\rangle$ $G:$ $\langle$ticket_out?, ent_update_dboard?$\rangle$ $T:$ δ(push_button!, ticket_out?) $= [0, 20) \wedge$ δ(take_ticket!, ent_update_dboard?) $= [0, 250]$
	Entry_Env_2	$A:$ $\langle$push_button!$\rangle$, $G:\langle$no_ticket_out?$\rangle$ $T:$ δ(push_button!, no_ticket_out?) $= [0, 20)$
	Entry_Env_3	$A:$ $\langle$push_button!, take_ticket!$\rangle$ $G:$ $\langle$ticket_out?, ent_time_out?$\rangle$ $T:$ δ(push_button!, ticket_out?) $= [0, 20)$
	Entry_Env_4	$A:\langle$initialize!$\rangle$, $G:\langle$reset_dboard?$\rangle$
Exit_Environment	Exit_Env_1	$A:$ $\langle$(ticket_insert!$\rangle$, $G:\langle$ticket_ok? $\prec$ ex_update_dboard?$\rangle$ $T:$ δ(ticket_ok?, ex_update_dboard?) $= [0, 250]$
	Exit_Env_2	$A:\langle$ticket_insert!$\rangle$, $G:\langle$ticket_error?$\rangle$
	Exit_Env_3	$A:\langle$ticket_insert!$\rangle$, $G:\langle$ticket_ok? $\preceq$ ex_time_out?$\rangle$

*All times are in nanoseconds.

δ(car_in?, ent_update_dboard!) = $[42, 142]$ in the Display_1 schedule of the Display component), but no time constraint between signals ent_update_dboard and push_button. Our solution was to add the time constraint to the environment assumptions.

Since we used AGR to verify VPMS, the above five errors were found using lesser CPU time and memory space, compared to that without using AGR. The first two and last errors were found by merely constructing a state-graph representing the concurrent behavior of Entry_Environment and Display, with a size of 54 modes and 159 transitions, which is much smaller compared to the total sizes of state-graphs constructed without AGR: 1375 modes and 4360 transitions. The third and forth errors were found by constructing the following two state-graphs: (1) Concurrent merge of Entry and Display: 14 modes, 17 transitions, and (2) Concurrent merge of Exit and Display: 251 modes, 636 transitions. All these state-graphs were much smaller in size compared to the total sizes when AGR was not used. This shows we can scale-up verification for complex systems and speed-up verification for simple systems. The tool used was *State-Graph Manipulators* (SGM) [27].

5 Conclusion

With the rapid progress of computer and electronic technology, guaranteeing the correctness of systems is no more easier than actually designing the system. For example, the verification of a System-on-Chip accounts for as much as 70% of the total design time. We need practical automatic techniques that can handle such highly complex systems. The work presented on assume-guarantee reasoning (AGR) in this article is one step towards that goal. Besides giving an algorithm for incorporating AGR into tools, we proposed an automatic generation procedure for assumptions, guarantees, and time constraints in real-time embedded systems. We quantified the advantages of applying AGR as against that without AGR. Our experiments on a fairly complex system such as the *Vehicle Parking Management System* corroborates our claims of the benefits obtained from applying AGR to invariant checking. Future research directions include applying AGR to other larger applications and integrating AGR techniques with informal validation techniques such as simulation and testing.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] M. Abadi and L. Lamport. Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.

[3] R. Alur, L. de Alfaro, R. Grosu, T. A. Henzinger, M. Kang, R. Majumdar, F. Mang, C. M. Kirsch, and B. Y. Wang. MOCHA: A model checking tool that exploits design structure. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 835–836, 2001.

[4] R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automating modular verification. In *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99), Lecture Notes in Computer Science 1664*, pages 82–97. Springer-Verlag, 1999.

[5] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183 – 235, 1994.

[6] R. Alur, R. Grosu, and B.-Y. Wang. Automated refinement checking for asynchronous processes. In *Proceedings of the 3rd International Conference on Formal Methods for Computer-Aided Design (FMCAD'00)*, 2000.

[7] R. Alur and T. A. Henzinger. Local liveness for compositional modeling of fair reactive systems. In P. Wolper, editor, *Proceedings of the International Conference on Computer-Aided Verification (CAV'95), Lecture Notes in Computer Science 939*, pages 166–179. Springer-Verlag, 1995.

[8] R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual Symposium of Logic in Computer Science (LICS'96)*, pages 207–218. IEEE CS Press, 1996.

[9] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: Modularity in model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'98), Lecture Notes in Computer Science 1427*, pages 521–525, 1998.

[10] A. T. Eiriksson. The formal design of 1M-gate ASICs. In G. Gopalakrishnan and P. Windley,

editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), Lecture Notes in Computer Science 1522*, pages 49–63, 1998.

[11] T. A. Henzinger, X. Liu, S. Qadeer, and S. K. Rajamani. Formal specification and verification of a dataflow processor array. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'99)*, pages 494–499, 1999.

[12] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee. In *Proceedings of the International Conference on Computer-Aided Verification (CAV'98), Lecture Notes in Computer Science 1427*, pages 440–451. Springer-Verlag, 1998.

[13] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Assume-guarantee refinement between different time scales. In N. Halbwachs and D. Peled, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'99), Lecture Notes in Computer Science 1633*, pages 208–221. Springer-Verlag, 1999.

[14] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'00)*, pages 245–252, 2000.

[15] T. A. Henzinger, S. Qadeer, S. K. Rajamani, and S. Tasiran. An assume-guarantee rule for checking simulation. In G. Gopalakrishnan and P. Windley, editors, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), Lecture Notes in Computer Science 1522*, pages 421–432. Springer-Verlag, 1998.

[16] P.-A. Hsiung. Embedded software verification in hardware-software codesign. *Journal of Systems Architecture — the Euromicro Journal*, 46(15):1435–1450, December 2000.

[17] P.-A. Hsiung. Hardware-software timing coverification of concurrent embedded real-time systems. *IEE Proceedings — Computers and Digital Techniques*, 147(2):81–90, March 2000.

[18] P.-A. Hsiung, S.-Y. Cheng, and T.-Y. Lee. Compositional verification of synchronous real-time embedded systems. In *Proc. of the 2002 VLSI Design/CAD Symposium (VLSI'02, Taitung, Taiwan)*, pages 187 – 190, August 2002.

[19] C. B. Jones. Tentative steps towards a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[20] T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen. DESC: A hardware-software codesign methodology for distributed embedded systems. *IEICE Transactions on Information and Systems*, E84-D(3):326–339, March 2001.

[21] T.-Y. Lee, P.-A. Hsiung, and S.-J. Chen. Hardware-software multi-level partitioning for distributed embedded multiprocessor systems. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E84-A(2):614–626, February 2001.

[22] K. L. McMillan. Verification of an implementation of tomasulo's algorithm by compositional model checking. In A. Hu and M. Vardi, editors, *Proceedings of the International Conference on Computer-Aided Verification (CAV'98), Lecture Notes in Computer Science 1427*, pages 110–121. Springer-Verlag, 1998.

[23] K. L. McMillan. Circular compositional reasoning about liveness. In L. Pierre and T. Kropf, editors, *Proceedings of the International Conference on Correct Hardware Design and Verification (CHARME'99), Lecture Notes in Computer Science 1703*, pages 342–345. Springer-Verlag, 1999.

[24] J. Misra and K. M. Chandy. Proofs of networks of processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

[25] S. K. Roy, H. Iwashita, and T. Nakata. Formal verification based on assume and guarantee approach – a case study. In *Proceedings of the Asia-Pacific Design Automation Conference (ASP-DAC'00)*, pages 77–80, 2000.

[26] N. Shankar. Lazy compositional verification. *Lecture Notes in Computer Science*, 1536, 1997.

[27] F. Wang and P.-A. Hsiung. Efficient and user-friendly verification. *IEEE Transactions on Computers*, 51(1):61 – 83, January 2002.

[28] M. Zulkernine and R. E. Seviora. Assume-guarantee supervisor for concurrent systems. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, pages 1552–1560, April 2001.