Assume-Guarantee Model Checking of Software: A Comparative Case Study *

Corina S. Păsăreanu, Matthew B. Dwyer, and Michael Huth

Department of Computing and Information Sciences
Kansas State University, Manhattan, KS 66506, USA
{pcorina,dwyer,huth}@cis.ksu.edu

Abstract. A variety of assume-guarantee model checking approaches have been proposed in the literature. In this paper, we describe several possible implementations of those approaches for checking properties of software components (*units*) using SPIN and SMV model checkers. Model checking software units requires, in general, the definition of an environment which establishes the run-time context in which the unit executes. We describe how implementations of such environments can be synthesized from specifications of assumed environment behavior written in LTL. Those environments can then be used to check properties that the software unit must guarantee which can be written in LTL or ACTL. We report on several experiments that provide evidence about the relative performance of the different assume-guarantee approaches.

1 Introduction

Model checking is maturing into an effective technique for validating and verifying properties of complex systems and is beginning to be included as part of system quality assurance activities. While most of the practical impact of model checking has been in the domains of hardware and communication protocols, the techniques and tools that support model checking are beginning to see some application to software systems.

It is well-known that software defects are less costly the earlier they are removed in the development process. Towards this end, a number of researchers have worked on applying model checking to artifacts that appear throughout the software life-cycle, such as requirements [2], architectures [23], designs [16], and source code [7]. Source code, of course, is not a monolithic entity that is developed at one time. Source code evolves over time with different components, or *units*, reaching maturity at different points. Software units come in many different types. In well-designed software, a unit defines a layer of functionality with a narrow interface and a mechanism for hiding the details of its implementation from direct external access. Units may be invoked by other units through the operations of their interface and they may in turn invoke operations of other

* This work was supported in part by NSF and DARPA under grants CCR-9633388, CCR-9703094, and CCR-9708184 and by NASA under grant NAG-02-1209.

units; Java classes or Ada packages or tasks are examples of units. Testing units in isolation involves the definition of program components that invoke the operations of the unit, a *driver*, and that implement the operations used by the unit, a *stub*, in such a way that the behavior of the unit is exercised in some desired fashion. Stubs and drivers can be defined to also represent *parallel contexts*. Parallel contexts represent those portions of an application that execute in parallel with and engage in inter-task communication with the procedures and tasks of the software unit under test. Standard practice in modern software development is to begin the process of unit testing as soon as each unit is "code-complete". While testing will remain an important part of any software development process, model checking has the potential to serve as an effective complement to testing techniques by detecting defects relative to specific correctness properties and in some cases verifying properties of the software.

In this paper, we describe our adaptation and application of assume-guarantee style model checking to reasoning about correctness properties of software units, written in Ada. Units are fundamentally *open* systems and must be *closed* with a definition of the environment that they will execute in. The software components used to achieve this environment definition serve the role of stubs and drivers. The naive environment for properties stated in universal logics is the *universal* environment, which is capable of invoking any sequence of operations in the unit's interface. In many cases, one has behavioral information about unit interfaces, rather than just signatures, that can be exploited to refine the definition of the environment used to complete the unit's definition. In particular, we use linear-temporal logic (LTL) [20] as a means of specifying assumptions about interface behavior. When both the assumption ϕ and the guarantee property ψ are specified in LTL one can simply check the formula $\phi \rightarrow \psi$ with a model checker like SPIN [16]. LTL assumptions can also be used to synthesize refined environments, in which case ϕ can be eliminated from the formula to be checked. An additional benefit of synthesizing such environments is that it enables guarantee properties (ψ) specified in the universal fragment of computation tree logic (CTL) [6] to be checked with a model checker like SMV [22].

The theoretical foundations of this approach are not new. Several researchers have explored the efficacy and complexity of different styles of assume-guarantee reasoning with LTL and CTL specifications [27, 19]. The primary contributions of this paper are pragmatic (*i*) implementing a tool to synthesize stubs and drivers that encode given LTL assumptions, (*ii*) supporting local assumptions about the behavior of individual components of the environment, and (*iii*) providing initial experimental evidence of the performance tradeoffs involved with different styles of assume-guarantee reasoning for software units using SPIN and SMV. Secondary contributions of the work presented here include preliminary examination of several "real" programs, including specifications and a discussion on how to analyze components of these programs and some preliminary data on the kinds of properties for which CTL model checking exhibits a performance advantage over LTL model checking. The paper proceeds by surveying relevant background material in the next section. Section 3 presents our procedure for

synthesizing Ada implementations of stubs and drivers from LTL assumptions. The Ada implementations are fed as input to an existing toolset for extracting finite-state models from source code which is described in Section 4. Section 5 then presents data on the performance of unit-level model checking based on synthesized environments. Section 6 describes related work and Section 7 concludes.

2 Background

Linear and Branching Temporal Logics. There are two principal types of temporal logics with discrete time: linear and branching. Linear temporal logic (LTL) is a language of assertions about computations. Its formulae are built from atomic propositions by means of Boolean connectives and the temporal connectives X ("next time") and U ("until"; $p\mathrm{U}q$ means that q holds at some point in the future, and that until that point, p is true). The formula $\mathrm{trueU}p$, abbreviated $\mathrm{F}p$, says that p holds *eventually*, and $\neg\mathrm{F}\neg p$, abbreviated $\mathrm{G}p$, says that p is *always* true. A program satisfies an LTL formula, if all its possible computations satisfy the formula. In contrast, computation tree logic (CTL) is a branching time logic about computation trees. Its temporal connectives consist of path quantifiers immediately followed by a single linear-temporal operator. The path quantifiers are A ("for all paths") and E ("for some path"). ACTL is the universal fragment of CTL. Using De Morgan's laws and dualities, any ACTL formula can be re-written to an equivalent CTL formula in which negations are applied only to atomic propositions, and that contains only A quantifiers; thus, in ACTL one can state properties of all computations of a program, but one can not state that certain computations exist.

Modular Verification. In modular verification, under the *assume-guarantee* paradigm [26], a specification consists of a pair $\langle\phi, \psi\rangle$, where ϕ and ψ are temporal logic formulae; ψ describes the guaranteed behavior of the module and ϕ describes the assumed behavior of the environment with which the module is interacting. For the linear temporal paradigm, both ϕ and ψ are LTL formulae. As observed in [26], in this case the assume-guarantee pair $\langle\phi, \psi\rangle$ can be combined to a single LTL formula $\phi \rightarrow \psi$. In the *linear branching modular model checking problem*, the assumption is an LTL formula, and the guarantee is a branching temporal logic formula (see e.g.[27]). Another approach is *branching modular model checking*, in which both assumption ϕ and guarantee ψ are branching temporal logic formulae. This case is considered in [14, 19]. In these papers it is argued that, in the context of modular verification, it is advantageous to use only *universal* temporal logic (like LTL and ACTL). Universal temporal logic formulae have the helpful property that once they are satisfied in a module, they are also satisfied in any system that contains the module. We consider in this paper assumptions expressed in LTL and guarantees that can be expressed in both LTL and ACTL.

SPIN and SMV. We use two finite-state verification tools: SPIN, a reachability based model checker that explicitly enumerates the state space of the system being checked; and SMV, a "symbolic model checker", which uses Ordered Binary Decision Diagrams to encode subsets of the state space. These tools represent two of the major approaches to finite-state verification. SPIN accepts design specifications written in the Promela language and it accepts correctness properties written in LTL. SPIN can be used for assume-guarantee style verification by checking LTL specifications of the form $\phi \rightarrow \psi$ against "closed" modules. SMV checks properties written in CTL, with *fairness* constraints of the form GFf, for some CTL formula f. Modular verification can be performed in a very limited way, if ϕ can be expressed via such fairness constraints. A new version of SMV supports LTL model checking, in addition to CTL, and it is also especially designed for assume-guarantee style reasoning, where both the assumption and the guarantee are LTL formulae.

3 Synthesis of Environments from LTL Assumptions

Tableau Procedure. We close a software unit by generating source code that implements models of environments. To begin construction of any such model one must have a definition of the possible actions of the environment. For Ada programs, these actions include: interface actions (i.e. entry calls or accepts, calls to interface procedures of the software unit) or some other internal actions of the environment. Based on this definition, we construct *universal* stubs and drivers that represent all possible sequences of actions. When LTL local assumptions are available, we can synthesize refined models of environments, using tableau-like methods [21, 13]. A *local assumption* describes the temporal relations assumed to exist among the executions of the interface operations of the unit, invoked by one particular environment.

We assume that the parameters in unit calls have been abstracted to finite domains; Section 4 discusses how this is achieved. We then use the algorithm from [13] (the same algorithm is used in SPIN for generating never claims) together with the subset construction, justified in [24], to construct from an LTL formula ϕ a *deterministic* automaton that can be represented as a graph (and translated to Ada). The graph is a *maximal* model [14] of the environment assumption in that every computation which satisfies the assumption is a path in the graph, and that every finite path in the graph is the prefix of some computation that satisfies the assumption ([13]). If ϕ is a *safety* assumption, then all the paths in the graph satisfy ϕ. But if ϕ is a *liveness* specification, then there exists some path in the graph that does not satisfy ϕ. In the case studies presented in this paper, we used only safety assumptions, and hence the verification using synthesized environments can not yield false negatives (i.e. negative results produced as a consequence of considering paths that do not conform with the environment assumptions). Synthesized environments can be used in model checking of (stutter-closed [1]) guarantees ψ written in LTL or ACTL. Note that the LTL assumptions are not necessarily stutter-closed.

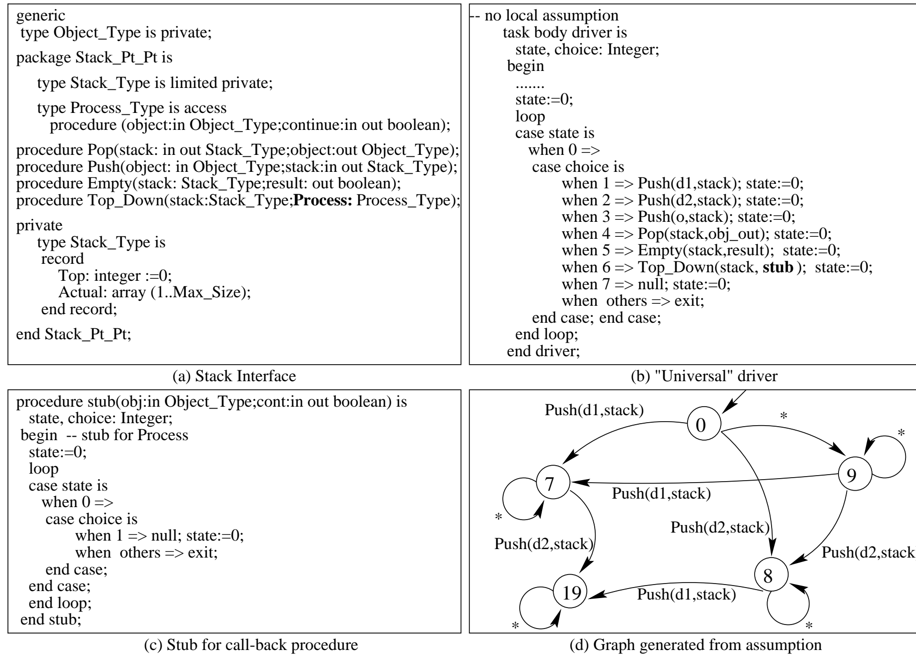


Fig. 1. Stack implementation

Our methodology for building maximal models of environments from local assumptions can be extended to handle global assumptions (i.e. assumptions that relate the behaviors of several local environments), as illustrated in the Replicated Workers Framework case study from Section 5.

An Example. To illustrate the techniques presented in this paper, we introduce a familiar software unit as an example. We consider the bounded stack implementation studied in [10] whose simplified interface is given in Figure 1(a). The implementation supports iteration in the stack-order (*Top_Down*), by invoking a user defined call-back routine (*Process*) for each datum stored in the stack. The universal driver is depicted in Figure 1(b); while the possible actions of the driver include all the interface operations of the stack package, we restrict the stub (presented in Figure 1(c)) to make no calls to the stack package. One property that is checked for this unit is that "If a pair of data are pushed then they must be popped in reverse order, provided that they are popped"(1s) (throughout the paper, we encode in the property names the systems they are referring to, e.g. (1s) means property 1 of the **s**tack). Checking this order-related property requires the notion of data-independence [28]. We abstracted variables of *Object_Type* using a 2-ordered data abstraction [9]. This abstraction maps two distinct values in the concrete domain to the tokens d_1 and d_2 and all other concrete values to o. As stated in [28], in order to generalize the results of such a restricted model check to all pairs of values of *Object_type* one must assure that the tokens are input to the system at most once, which is specified as the LTL

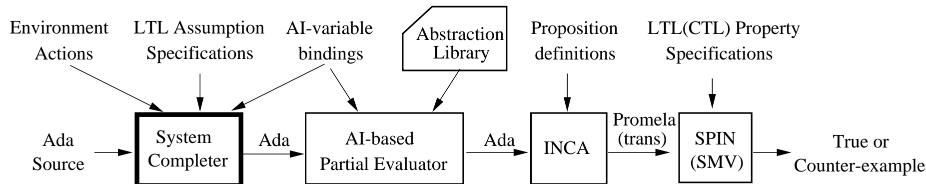


Fig. 2. Source Model Extraction Tools

assumption (about the driver):
(G(Push(d_1,stack)→ XG¬Push(d_1,stack)))∧G(Push(d_2,stack) →XG¬Push(d_2,stack)).
In Figure 1(d) we show the graph generated from the above assumption, where by $*$, we denote any other possible action of the driver: *Push(o,stack)*, *Pop(stack, obj_out)*, *Empty(stack,result)*, *Top_Down(stack,stub)* and *null*. In the Ada code, the internal non-observable actions of the environment are modeled by the *null* statement, internal choice in the environment is modeled by using the *choice* variable which will be abstracted to the *point* abstraction [9], and finally the abstracted parameter values are enumerated in specialized *Push* calls.

4 A Model Extraction Toolset

Toolset Components. For model checking properties of the Ada programs described in Section 5, we apply the methodology described in [9, 11] which is supported by the tool set illustrated in Figure 2. The first tool component constructs source code for drivers and stubs that complete the software unit, as discussed in Section 3. Incompletely defined Ada source code is fed to the System Completer, together with a description of the possible actions of the local environments and the LTL local assumptions written in terms of the environment actions; the names of the actions are uninterpreted strings and subsequent compile checks are performed for correctness. The completed program is abstracted and simplified using abstract interpretation [8] and partial evaluation [18] techniques. The resulting finite-state Ada program can be compiled by the INCA tool set [7] into the input languages of several verification tools. An important feature of INCA is its support for defining propositions for the observable states and actions of the system that the user is interested in reasoning about. These propositions have provided an advantage for interpreting counter examples, since the counter example will be rendered in terms of those states and actions.

The Stack Example. For the stack package presented in Section 3, we wrote the LTL and ACTL specifications derived from the English language description of the software package. After we closed the stack unit with a stub and a driver, we defined predicates *callPush_*d_1 and *callPush_*d_2 that are true immediately after the driver calls interface operation *Push* with the first parameter set to d_1 and d_2 respectively. Analogously, predicates *returnPop_*d_1 and *returnPop_*d_2 are true immediately after operation *Pop* called by the driver returns with *obj_out* set to d_1 and d_2 respectively. Using these predicates, property (1s) can be specified

in the following way:
(1s) **If d_1 and d_2 are pushed in this order, then d_1 will not be popped until d_2 is popped or d_1 will never be popped.**
LTL: G((callPush_d_1 $\wedge$ ($\neg$returnPop_d_1 U callPush_d_2)) $\rightarrow$
($\neg$returnPop_d_1 U (returnPop_d_2 $\vee$ G $\neg$returnPop_d_1)))
ACTL: $\neg$EF(callPush_d_1 $\wedge$ (E($\neg$returnPop_d_1 U (callPush_d_2 $\wedge$
E($\neg$returnPop_d_2 U (returnPop_d_1 $\wedge$ $\neg$returnPop_d_2))))))

Special control points can be added to the finite-state model of the Ada program; for example, we added such a control point immediately after procedure *Pop* returns d_1 and we defined the predicate *after_returnPop_*d_1, which is true just after predicate *returnPop_*d_1 becomes false. We used this predicate for specifing a new stack property:
(2s) **Once d_1 is popped, it can not be popped again.**
LTL: G(after_returnPop_d_1 $\rightarrow$ G $\neg$returnPop_d_1)
ACTL: AG (after_returnPop_d_1 $\rightarrow$ AG $\neg$returnPop_d_1)

We can also use predicates that define the points at which selected program variables hold a given value. In the stack package, for example, we can define the predicate *TopEQzero* which holds in the states where variable *Top* is zero.

5 Experiments with Synthesized Environments

To assess the potential benefits of using synthesized environments in assume-guarantee model checking of software units, we analyzed several components of software systems. All of the properties we checked (a selection of which is given in Figure 3) are instances of property specification patterns [12]. In this section, we begin with brief descriptions of the software systems we analyzed. In Figure 4 depicting system architectures, lines with arrows represent either procedure invocations or calls to task entries; little shaded rectangles represent interface operations for software units. We then compare the times for SPIN and SMV model checking with universal and synthesized environments. Space limits prohibit the inclusion of all of the details of these studies, but, we have collected the original Ada source code, synthesized environment components, abstracted finite-state Ada code, proposition definitions, assumptions, properties, and Promela and SMV input descriptions on a web-site [25].

The Gas Station (g). The problem was analyzed in [4] using SPIN and SMV; it is a scalable concurrent simulation of an automated gas station. Its architecture is depicted in Figure 4(a). We analyzed the server subsystem, which consists of operator and pump processes that maintain a bounded length queue holding customers' requests. The environment consists of the customer tasks. We checked property (1g), for three versions of the gas station: with two (version g_2), three (version g_3) and four (version g_4) customers, respectively. Model checking the property on the server subsystem "closed" with the universal environment yields a counter example in which customer 1 makes a prepayment while using the pump, and thus it keeps using the pump indefinitely. We then assumed that a

(1g) If customer 2 prepays while customer 1 is using the pump then the operator will activate the pump for customer 2 next.

LTL: G((Start1 ∧ (¬Stop1 U Prepay2)) →
(¬Activate1 U (Activate2 ∨ G ¬Activate1)))

ACTL: ¬EF(Start1 ∧ E(¬Stop1 U (Prepay2 ∧
E(¬Activate2 U (Activate1 ∧ ¬Activate2)))))

(4r) The computation does not terminate unless the pool is empty (variable *workCount* is zero), or a worker signals work is *done*.

LTL: G(callExecute → (¬returnExecute U
(done ∨ workCountEQzero ∨ G ¬returnExecute)))

ACTL: AG(callExecute → ¬E(¬(done ∨ workCountEQzero) U
(returnExecute ∧ ¬(done ∨ workCountEQzero))))

(5r) If a worker is ready to *Get* work, the workpool is not empty and the computation is not *done*, then eventually work is scheduled.

LTL: G((workCountGRzero ∧ acceptPoolGet ∧ ¬done)→
F(calldoWork1 ∨ calldoWork2 ∨ calldoWork3))

ACTL: AG((workCountGRzero ∧ acceptPoolGet ∧ ¬done)→
AF(calldoWork1 ∨ calldoWork2 ∨ calldoWork3))

(6r) The work pool schedules work in input order.

LTL: G((returnInput_d_1 ∧ F returnInput_d_2)→
(¬callGet_d_2 U (callGet_d_1 ∨ G ¬callGet_d$_2$)))

ACTL: ¬EF(returnInput_d_1 ∧ E(¬callGet_d_1 U
(returnInput_d_2 ∧ E(¬callGet_d_1 U (callGet_d_2 ∧ ¬callGet_d_1)))))

(7r) If stub *doWork* is invoked by worker task i on item d_1 then no other worker task j will invoke *doWork* on the same item d_1.

LTL: G(calldoWorki_d_1 → G ¬calldoWorkj_d_1))

ACTL: AG(calldoWorki_d_1 → AG ¬calldoWorkj_d_1)

(8r) If worker task i invokes *doWork* on d_1, that same worker task will not invoke *doWork* on d_1 again.

LTL: G(returndoWorki_d_1 → G ¬calldoWorki_d_1)

ACTL: AG(returndoWorki_d_1 → AG ¬calldoWorki_d_1)

(7c) If artist a_1 registers for event e_1 before artist a_2 does, then (until unregistration or termination) once dispatcher receives event e_1 from ADT it will not notify a_2 before notifying a_1.

LTL: G((register_a_1e_1 ∧ (¬(unregister_a_1e_1 ∨ unregister_a_2e_1) U
register_a_2e_1) ∧ F(term ∨ unregister_a_1e_1 ∨ unregister_a_2e_1))→
((notify_artists_e_1 → (¬notify_client_a_2e_1 U(notify_client_a_1e_1 ∨
G¬notify_client_a_2e_1)))U(term ∨ unregister_a_1e_1 ∨ unregister_a_2e_1)))

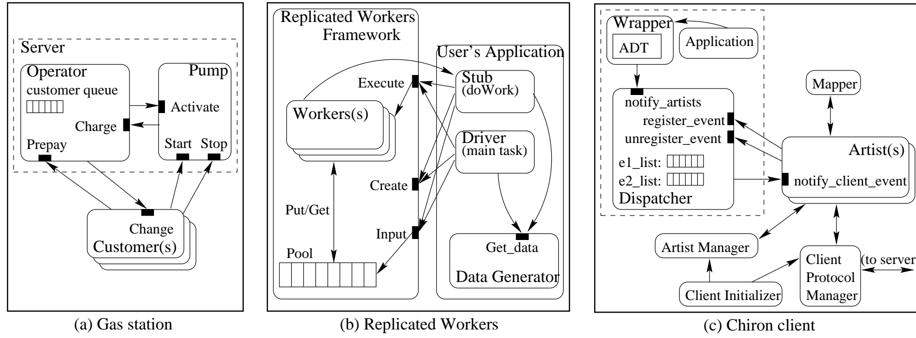
ACTL: ¬EF(register_a_1e_1 ∧ E((¬unregister_a_1e_1 ∧ ¬notify_client_a_1e_1) U
(register_a_2e_1 ∧ E((¬unregister_a_1e_1 ∧ ¬unregister_a_2e_1 ∧
¬notify_client_a_1e_1) U notify_client_a_2e_1))))

(8c) No artist attempts to register for event e_1 when the size of the array that stores artists registered for event e_1 is equal to the number of artists.

LTL: G(e1szEQ2 ∧ (after_register_a_1e_1 ∨ after_register_a_2e_1) →
(¬(register_a_2e_1 ∨ register_a_1e_1) U
(e1szLT2 ∨ G ¬(register_a_2e_1 ∨ register_a_1e_1))))

ACTL: AG(e1szEQ2 ∧ (after_register_a_1e_1 ∨ after_register_a_2e_1) →
¬E(¬e1szLT2 U ((register_a_1e_1 ∨ register_a_2e_1) ∧ ¬e1szLT2)))

Fig. 3. Specifications



(a) Gas station (b) Replicated Workers (c) Chiron client

Fig. 4. Software architectures

customer does not prepay for subsequent pumping during current pumping. The LTL assumption for customer 1 is:
G(Pump.Start →(¬Operator.Prepay(1) U (Pump.Stop ∨ G ¬Operator.Prepay(1)))).

Model checking property (1g) with the behavior of customer 1 restricted by the environment assumption yields true results.

The Replicated Workers Framework (r). This is a parameterizable job scheduler implemented as an Ada generic package which manages the startup, shutdown, and interaction of processes internally. The environment consists of a single driver (the main task) and two kinds of stub routines (*doWork* and *doResult*). Figure 4(b) illustrates a slightly simplified structure of the replicated workers framework and its interaction with the environment (the user's application). A variety of properties of this system (with three worker tasks) were model checked in [9], using SPIN. We reproduce here only the properties that needed some environment assumptions.

The environment assumptions used for checking both properties (4r) and (5r) restrict the stubs for the user-defined procedures to make no calls to the replicated workers framework: G ¬(Create ∨ Input ∨ Execute).

For properties (6r), (7r) and (8r) we used the 2-ordered data abstraction described in Section 3. As discussed in Section 3, properties like (6r), (7r) and (8r) are true only under the environment assumption that d_1 and d_2 are input to the system only once. We observe that the latter assumption is global, rather than local, because work items can be input to the system from different tasks/environments (passed in from the main task and returned from calls to the *doWork* stubs). To enforce the necessary global assumption we created a new task, *Data_Generator*, that generates data items d_1, d_2 and o. The driver and the stubs (implemented as universal environments) communicate with the *Data_Generator* to *Get_data* to be input in the work pool. The LTL assumption used in the synthesis of *Data_Generator* is:
$(\text{G}(\text{Get_data}(d_1) \rightarrow \text{XG}\neg\text{Get_data}(d_1))) \wedge \text{G}(\text{Get_data}(d_2) \rightarrow \text{XG}\neg\text{Get_data}(d_2))$.

The Chiron Client (c). Chiron [29] is a user interface development system. The application for which an interface is to be constructed must be organized as abstract data types (ADTs). Chiron has a client-server architecture; Figure 4(c)

gives a simplified view of the architecture of a client. The *Artists* maintain the graphical depictions of the ADTs. They indicate the events they are interested in by registering and de-registering with the *Dispatcher*, which notifies the artists of the events from the ADTs. We analyzed the dispatcher subsystem, which consists of the dispatcher, the wrapper and the application; while the environment consists of the artists and the other remaining components. We checked several properties for two versions of the dispatcher subsystem: with two artists registering for two events (version c_2) and with two artists registering for three events (version c_3), respectively. Only two properties required the use of environment assumptions: (7c) and (8c). Property (8c) can be written as a conjunction of two properties of the following form, that refer to only one artist a_i:

(8ic) **No artist a_i attempts to register for event e_1 when the size of the array that stores artists registered for event e_1 is equal to the number of artists.**

LTL:G(e1szEQ2 $\wedge$ after_register_$a_i e_1$ $\rightarrow$($\neg$register_$a_i e_1$ U(e1szLT2 $\vee$G$\neg$register_$a_i e_1$)))

ACTL:AG(e1szEQ2 $\wedge$after_register_$a_i e_1$$\rightarrow$$\neg$E($\neg$e1szLT2 U(register_$a_i e_1$ $\wedge$$\neg$e1szLT2)))

When model checking the properties on the dispatcher subsystem closed with universal environments, we obtained false results and counter examples in which one of the artists keeps registering for event e_1 while registered for event e_1. We used the assumption that an artist never registers for an event if it is already registered for that event, which is actually a specification of the Chiron system. The LTL assumption for artist a_1 is:

G(dispatcher.register(a_1,e_1) $\rightarrow$ X(($\neg$dispatcher.register(a_1,e_1)) U
(dispatcher.unregister(a_1,e_1) $\vee$ G$\neg$dispatcher.register(a_1,e_1)))).

We obtained true results when model checking with both artists a_1 and a_2 restricted by the environment assumptions.

Generic Containers Implementations. In [10] a large collection of properties were checked of implementations of bounded **stack (s)**, **queue (q)** and **priority queue (p)**. These are sequential software units that were completed with a single driver environment component and with stubs for user-defined procedures. We already described the stack implementation in Section 3. Similar properties were checked for queue and priority queue.

Results. In each of these studies, several of the properties required environment assumptions for successful model checking. These model checks were performed with synthesized environments and, alternatively, with assumptions in the formulae (for LTL); we used SPIN, version 3.09 and SMV (Cadence version), on a SUN ULTRA5 with a 270Mhz UltraSparc IIi and 128Meg of RAM.

Figure 5 gives the data for each of the model checking runs using SPIN. The names in the first column encode the systems and property that are checked; a subscript denotes the version of the system being analyzed. For model checking specifications of the form $\phi \rightarrow \psi$ using universal environments, we report the total of user and system time in seconds to convert LTL to the SPIN input format, i.e. never claim, (t_{never}) and to execute the model checker (t_{MC}). We also report the memory used in verification in Mbytes (mem) and the total time to construct Promela code from the initial Ada program (t_{build}). For the

$Prop.$	t_{never}	t_{MC}	mem	t_{build}	t'_{never}	t'_{MC}	mem'	t'_{build}
$1g_2$	655.5	3.8	2.005	1.6	0.1	0.5	1.698	1.6
$1g_3$	655.5	149.4	11.938	1.7	0.1	18.4	4.565	1.8
$1g_4$	655.5	5604.7	268.348	2.6	0.1	606.5	101.436	2.7
$4r$	0.3	7.4	5.385	11.9	0.1	3.1	2.620	7.6
$5r$	0.1	21.1	2.313	11.9	0.1	12.4	2.005	7.6
$6r$	539.5	486.4	77.372	36.8	0.1	330.1	66.415	47.2
$7r$	0.1	567.2	78.089	36.8	0.1	242.7	51.669	47.2
$8r$	0.1	548.4	73.481	36.8	0.1	227.1	48.495	47.2
$7c_2$	-				25.1	30.4	4.975	6.5
$81c_2$	63.0	223.1	9.788	6.1	0.1	21.6	3.746	6.5
$82c_2$	62.6	236.9	10.095	6.1	0.1	22.1	3.746	6.5
$7c_3$	-				25.1	132.8	16.137	11.8
$81c_3$	63.0	700.3	29.551	7.4	0.1	114.6	10.812	11.8
$82c_3$	62.6	1348.1	33.033	7.4	0.1	119.9	10.812	11.8
$1s$	367.5	0.2	2.108	27.1	0.2	0.1	2.005	15.5
$2s$	0.1	0.2	2.005	27.1	0.1	0.1	2.005	15.5
$1p$	338.2	0.2	2.517	35.3	0.1	0.1	2.108	25.6
$2p$	365.6	0.3	2.517	35.3	0.1	0.1	2.108	25.6
$3p$	0.1	0.3	2.313	35.3	0.1	0.1	2.108	25.6
$1q$	513.9	0.1	2.005	27.8	0.1	0.1	1.801	22.4
$2q$	0.1	0.1	1.903	27.8	0.1	0.1	1.801	22.4

Fig. 5. SPIN results

verification using synthesized environments we report measurements of the same times and sizes; a $'$ denotes the synthesized measure. SPIN ran out of memory, when we tried to generate never claims for properties (7c) and (8c), with the assumptions encoded in the formulae of the form $\phi_{a1} \wedge \phi_{a2} \rightarrow property$, where ϕ_{a1} and ϕ_{a2} are the local assumptions about artists a_1 and a_2. However, we were able to generate never claims for properties (81c) and (82c).

Figure 6 gives the data for each of the model checking runs using SMV. For verifying specifications using universal environments when both ϕ and ψ are LTL formulae, we report the total of user and system time in seconds to model check specifications (t_{MC_LTL}), the memory used in Mbyte (mem_{LTL}) and the total time to construct SMV input files from the Ada program (t_{build}). For model checks using synthesized environments, we report the measurements of the same times and sizes (denoted by $'$). In addition, synthesized environments can be used in CTL model checking, for which we give the total model checking time (t'_{MC_CTL}) and the memory used (mem'_{CTL}).

Discussion. Our data indicate that synthesized environments enable faster model-checking. Figure 7 plots the ratio of model check time (memory) for a property ψ relative to checks of $\phi \rightarrow \psi$ with SPIN using the universal environment (i.e t'_{MC}/t_{MC} and mem'/mem). In all cases, the synthesized environments enabled reductions in model check times and memory. The performance advantage arises from the fact that while the universal environment is smaller than the synthesized environment in most cases, the assumption must be encoded in the formulae to be checked and this contributes, in general, as many states to the never claim as to the synthesized environment. This leads to larger state spaces to be searched. We note that in two cases it was impossible to generate never claims from LTL formulae (of the form $\phi \rightarrow \psi$), while the synthesized environments made model checking possible.

$Prop.$	t_{MC_LTL}	mem_{LTL}	t_{build}	t'_{MC_LTL}	mem'_{LTL}	t'_{MC_CTL}	mem'_{CTL}	t'_{build}
$1g_2$	5.56	4.898	3.9	2.34	4.505	0.78	4.071	5.1
$1g_3$	23.05	6.913	8.9	8.03	6.274	2.77	5.488	11.5
$1g_4$	127.49	18.136	31.8	52.00	13.844	16.13	11.517	39.6
$4r$	67.23	23.478	158.14	16.3	9.379	11.03	8.339	55.69
$5r$	95.31	32.112	158.14	26.10	12.574	25.98	12.631	55.69
$6r$	611.09	28.442	130.4	430.73	30.113	63.27	20.922	114.12
$7r$	74.44	20.160	130.4	60.18	20.922	62.66	20.922	114.12
$8r$	74.16	20.160	130.4	60.53	20.922	62.39	20.922	114.12
$7c_2$	266.91	18.751	9.3	90.66	28.122	56.06	24.674	15.41
$8c_2$	51.6	13.475	9.3	59.81	24.870	55.69	24.543	15.41
$7c_3$	593.74	43.704	19.1	382.71	73.449	283.6	67.346	44.8
$8c_3$	191.18	32.235	19.1	161.14	47.054	157.28	45.907	44.8
$1s$	3.42	5.627	40.5	2.14	4.939	0.88	3.399	25.5
$2s$	2.21	5.332	40.5	0.8	3.399	0.85	3.399	25.5
$1p$	11.49	8.724	1:01.8	4.1	6.274	4.18	6.045	41.2
$2p$	11.16	8.716	1:01.8	4.80	6.266	4.13	6.045	41.2
$3p$	9.72	8.421	1:01.8	4.05	6.045	4.10	6.045	41.2
$1q$	3.09	5.398	40.2	2.10	4.874	1.60	4.792	41.2
$2q$	2.23	5.250	40.2	1.63	4.792	1.64	4.792	41.2

Fig. 6. SMV results

CTL model checking these systems with SMV was not feasible, previously, due to the difficulty of expressing assumptions in CTL. With synthesized environments, however, we can compare performances of LTL versus CTL model checking using SMV. Figure 8 plots the ratio of model check time (memory) for SMV using synthesized environments relative to universal environments. For each problem, the first bar gives the ratio for LTL model checking using SMV with the synthesized environment versus the universal SMV baseline (i.e. t'_{MC_LTL}/t_{MC_LTL} and mem'_{LTL}/mem_{LTL}) and the second bar gives the ratio for CTL model checking using SMV with the synthesized environment versus the universal SMV baseline (i.e. t'_{MC_CTL}/t_{MC_LTL} and mem'_{CTL}/mem_{LTL}). As in the case of SPIN, synthesized environments seem to enable faster model checking with SMV. In terms of memory requirements, it is not clear that synthesized environments are better. Further empirical study is needed to determine the kinds of assumptions to which the use of synthesized environments is suited to. The present data suggest to synthesize environments from assumptions that reduce the state space of the model (e.g. the assumption about the environment of the stack unit: "d_1 and d_2 are input to the stack only once").

The universal environment approach has the advantage of generality. In particular, the model generated with the universal environment can be reused with different assumptions, whereas synthesized environments encode a single set of assumptions. Thus for the universal case, model construction time could be amortized across a number of model checks. The data indicates, however, that model construction time is not the dominant factor in the overall analysis time and that the time to regenerate synthesized environments appears to be more than recovered by reduced model check times. This observation, however, is based on studying systems with relatively few external calls and a small interface action set. Also, we used relatively simple assumptions for which the synthesized environments were small (maximum 5 nodes in the generated graphs).

It is well-known that the complexity of model checking varies with the logic used, for example CTL model checking is linear in the size of the formula and

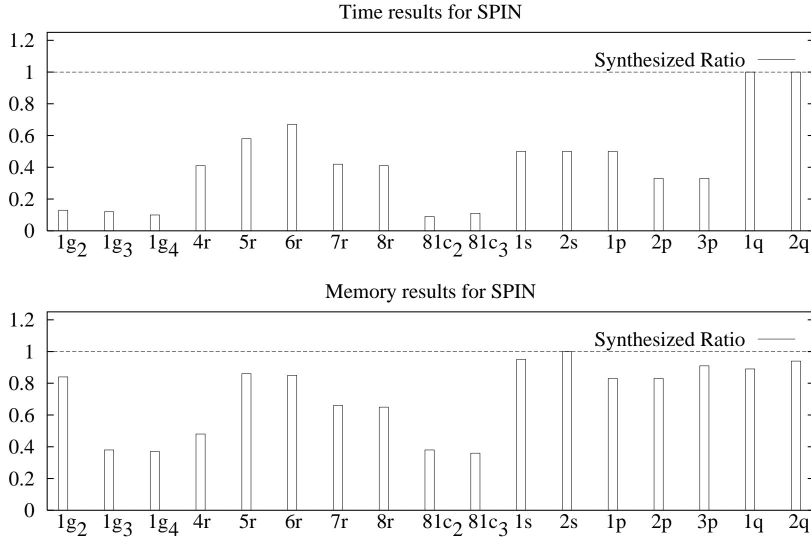


Fig. 7. Performance comparisons for SPIN model checking

LTL model checking is exponential. For practitioners, the relevant question is whether this distinction occurs in practice and if so how much more costly LTL model checking is. Since SMV supports both LTL and CTL model checking the data in Figure 6 (in columns t'_{MC_LTL} and t'_{MC_CTL}) can shed some light on the kinds of properties for which LTL model checking is more expensive than CTL model checking. One of the main problems in such a comparison is ensuring that the property specifications in the two logics are the same; we minimized this bias by using the predefined specification templates provided with the specification patterns system [12]. Ignoring scaling of systems and variations of encoded assumptions, there were a total of 15 properties checked. Of these, eight were faster to check in the CTL case, five were faster in the LTL case, and the other two took essentially the same time. In only four cases (1g,6r,7c,1s), was there a significant difference in model check time; each of these cases favored CTL. These four specifications are instances of the *global response-chain* pattern [12]. While it is tempting to conclude that CTL is advantageous for this class of specifications we observe that the LTL specifications for response-chains are not claimed to be "optimal" in any sense. A much broader study of the relationship between model check time, property being checked, and formulation of the property in temporal logics is needed to characterize the practical differences between LTL and CTL model checking. Our current results suggest that in most cases the difference is negligible but that certain forms of specifications may lend themselves to more efficient CTL model checking.

While not explicit in the data reported in this paper, it is interesting to note that in many cases, properties of the systems we studied could be model checked without any assumptions and, when necessary, relatively few assumptions were sufficient to achieve the level of precision necessary for property verification (out of 39 properties, only 18 properties needed assumptions).

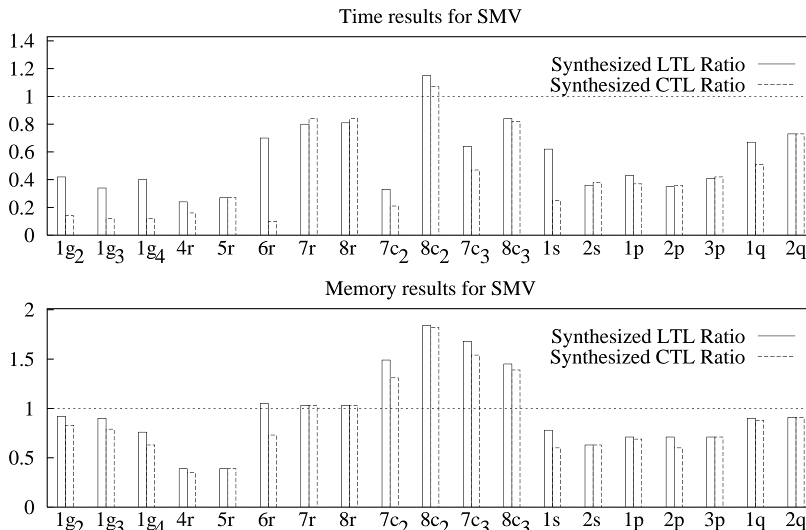


Fig. 8. Performance comparisons for SMV model checking

6 Related Work

The work described in this paper touches on model checking software systems and model checking open systems.

Much of the related work was described in Section 2. There has been some recent work on developing translation tools to convert software written in high-level programming languages to the input languages of model checkers. In addition to the INCA tools, which support Ada, there are two toolsets for translating Java programs to Promela, the input language of SPIN. JCAT [17] handles a significantly restricted subset of the Java language and Java Path Finder [15] handles a much larger portion of the language, including exceptions and inheritance. Neither of these tools provides any support for abstracting the control and data states of the program. We are working to port the environment synthesis tool described in this paper to generate environments in Java.

Our use of assumptions to synthesize a model of the environment is similar to work on compositional analysis. These divide-and-conquer approaches decompose a system into sub-systems, derive interfaces that summarize the behavior of each subsystem (e.g. [5]), then perform analyses using interfaces in place of the details of the sub-systems. This notion of capturing environment behavior with interfaces also appears in recent developments on theoretical issues related to modular verification (e.g. [20, 19]). There has been considerably less work on the practical issues involved with finite-state verification of partial software systems. Aside from our work reported in [9, 10], there is another recent related practical effort. Avrunin, Dillon and Corbett [3] have developed a technique that allows partial systems to be described in a mixture of source code and specifications. In their work, specifications can be thought of as assumptions on a naive completion

of a partial system given in code. Unlike our work, their approach is targeted to automated analysis of timing properties of systems.

7 Conclusion

We have presented an approach to model checking properties of software units *in isolation.* This approach is based on the synthesis of environments that encode LTL assumptions. The approach also enables LTL-ACTL assume-guarantee model checking. The reader should take care in making any direct comparison of the effectiveness of SMV and SPIN for support of assume-guarantee model checking. Such comparison would require a much broader study that carefully assesses the biases introduced by translating Ada to the model checker inputs as was done in [7]. The evidence seems to be conclusive on the question of whether assumptions should be encoded in the state space, i.e., environment, or the formula to be checked. For both LTL-LTL and LTL-ACTL approaches assume-guarantee model checking is more efficient with respect to time for safety assumptions encoded in the state space. This result holds regardless of whether the model checks are performed with SMV or SPIN.

References

1. M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73-132, January 1993.
2. J. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24-40, June 1993.
3. G.S. Avrunin, J.C. Corbett, and L.K. Dillon. Analyzing partially-implemented real-time systems. In *Proceedings of the 19th International Conference on Software Engineering*, May 1997.
4. A.T. Chamillard. *An Empirical Comparison of Static Concurrency Analysis Techniques.* PhD thesis, University of Massachusetts at Amherst, May 1996.
5. S.C. Cheung and J. Kramer. Checking subsystem safety properties in compositional reachability analysis. In *Proceedings of the 18th International Conference on Software Engineering*, Berlin, March 1996.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244-263, April 1986.
7. J.C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3), March 1996.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238-252, 1977.
9. M.B. Dwyer and C.S. Păsăreanu. Filter-based model checking of partial systems. In *Proceedings of the Sixth ACM SIGSOFT Symposium on Foundations of Software Engineering*, November 1998.
10. M.B. Dwyer and C.S. Păsăreanu. Model checking generic container implementations. In *Generic Programing: Proceedings of a Dagstuhl Seminar*, Lecture Notes in Computer Science, Dagstuhl Castle, Germany, 1998. to appear.

11. M.B. Dwyer, C.S. Păsăreanu, and J.C. Corbett. Translating ada programs for model checking : A tutorial. Technical Report 98-12, Kansas State University, Department of Computing and Information Sciences, 1998.
12. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.
13. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. Simple On-the-fly Automatic Verification of Linear Temporal Logic. In *Proceedings of PSTV'95*, 1995.
14. O. Grumberg and D.E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843-871, May 1994.
15. K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 1999. to appear.
16. G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279-294, May 1997.
17. R. Iosef. A concurrency analysis tool for java programs. Master's thesis, Polytechnic University of Turin, August 1997.
18. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.
19. O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking (extended abstract). In Insup Lee and Scott A. Smolka, editors, *CONCUR '95: Concurrency Theory, 6th International Conference*, volume 962 of *Lecture Notes in Computer Science*, pages 408-422, Philadelphia, Pennsylvania, 21-24 August 1995. Springer-Verlag.
20. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1991.
21. Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(1):68-93, 1984.
22. K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.
23. G.N. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying static analysis to software architectures. In *LNCS 1301.* The 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 1997.
24. C.S. Păsăreanu, M.B. Dwyer, and M. Huth. Modular Verification of Software Units. Technical Report 98-15, Kansas State University, Department of Computing and Information Sciences, 1998.
25. C.S. Păsăreanu and M.B. Dwyer. Software Model Checking Case Studies. `http://www.cis.ksu.edu/santos/bandera/index.html#case-studies`, 1998.
26. A. Pnueli. In transition from global to modular temporal reasoning about programs. In K. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123-144. Springer-Verlag, 1985.
27. M.Y. Vardi. On the complexity of modular model checking. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 101-111, San Diego, California, 26-29 June 1995. IEEE Computer Society Press.
28. P. Wolper. Specifying interesting properties of programs in propositional temporal logics. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 184-193, St. Petersburg, Fla., January 1986.
29. M. Young, R.N. Taylor, D.L. Levine, K.A. Nies, and D. Brodbeck. A concurrency analysis tool suite: Rationale, design, and preliminary experience. *ACM Transactions on Software Engineering and Methodology*, 4(1):64-106, January 1995.