11-2010

Model checking a model checker: A code contract combined approach

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Bin CHENG

Citation

SUN, Jun; LIU, Yang; and CHENG, Bin. Model checking a model checker: A code contract combined approach. (2010). *Proceedings of the 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19*. 518-533.
Available at: https://ink.library.smu.edu.sg/sis_research/5036

Model Checking a Model Checker: A Code Contract Combined Approach*

Jun Sun[1], Yang Liu[2], and Bin Cheng[2]

[1] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] School of Computing, National University of Singapore
{liuyang,chenbing}@comp.nus.edu.sg

Abstract. Model checkers, like any complex software, are subject to bugs. Unlike ordinary software, model checkers are often used to verify safety critical systems. Their correctness is thus vital. Verifying model checkers is extremely challenging because they are always complicated in logic and highly optimized. In this work, we propose a code contract combined approach for checking model checkers and apply it to a home-grown model checker PAT. In this approach, we firstly embed programming contracts (i.e., pre/post-conditions and invariants) into its source code, which can capture correctness of model checking algorithms, underlying data structures, consistency between different model checking parameters, etc. Then, interface models of complicated data structures and graphical user interfaces (GUI) are built and model checked. By linking the interface models with actual source codes and exhausting all execution sequences of interface models using PAT, *we model check PAT using itself*! Our experience shows that the approach is effective in identifying common bugs or subtle flaws that result from extremely improbable events.

1 Introduction

After two decades of development, model checking [8] has emerged as an effective method for verification of critical systems. It has established as a system validation method complementing standard techniques like simulation and testing. There have been a number of recent successful stories. The static driver verifier which uses the SLAM verification engine has been reported to find many driver model violations [1]. In 2009, it was reported that model checking has been used to replace testing in Intel CoreTM i7 processor (with millions of registers) execution engine validation [17].

Model checkers, like any non-trivial software, are subject to bugs. This is evidenced by the bug collection for established model checkers like SPIN [14] and NuSMV [7]. Model checkers are, nonetheless, distinguished from ordinary software due to their very nature. Firstly, they are always complicated in computational logic. Many complicated model checking algorithms, in the name of efficiency, have been proposed to verify a variety of system properties. Furthermore, sophisticated state reduction techniques are

* This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 518–533, 2010.

often applied, for instance, partial order reduction [22], symmetric reduction [10], data abstraction [2], or their combinations. Secondly, because efficiency is essential, model checkers are often highly optimized, which implies that they may not be designed for rigorous system maintenance or testing. Lastly, because model checking techniques are developing rapidly, model checkers are often updated frequently. This is evidenced by the update history of popular model checkers like SPIN, Uppaal, and so on.

Model checkers are often applied to safety critical systems. If a property is falsified, correctness of the model checker can be validated by checking whether the counterexample is a real one. It is when a property is claimed true - which is often the last act of formal verification, the correctness of the model checker must be assumed in order to trust the verification result. Given the importance of model checkers, it is essential to verify them formally or at least develop ways to systematically improve their quality.

There are multiple candidate approaches. First, theorem proving can be used to prove the correctness of model checking *algorithms*. It is, however, unpractical in verifying model checkers. The second candidate is push-button software verification technique like model checking. Still, completely verifying model checkers is beyond the capability of state-of-art model checking solutions. One important factor is their size. For instance, NuSMV has about 180 KLOC and SPIN has more than 30 KLOC. The current software verification tools can handle programs (often from constrained scenarios) up to tens of KLOC and often require manual simplifications of the code under analysis [21]. Furthermore, state-of-art software verification (e.g., the SLAM project [1]) relies on building an abstract finite state model from a program using predicate abstraction, which is highly non-trivial and expensive. Given that model checkers are often updated frequently, this approach is unpractical.

Contribution. In this work, we take the challenge of systematically validating a real-world model checker PAT [27]. PAT is a self-contained framework for system modeling, simulation and verification. After years of development, PAT has more than 600 KLOC, thousands of test cases as well as a list of discovered bugs. In order to systematically improve PAT's quality, we propose an approach which combines code contracts with model checking techniques. The contracts serve a correctness specification and model checking is used as an effective technique to search for contract violations as well as violations of additional critical properties. Formally developing and validating a formal verification system itself is the fundamental approach to increase user's confidence in the formal tools like model checkers. Though we cannot completely verify PAT, the approach is effective and scalable.

Approach. After evaluating our options, the following approach is developed. Firstly, given that PAT is developed using C# in .NET, we make use of the code contracts project [4] and systematically express coding assumptions in the source codes. The code contracts take the form of object invariants, method precondition and post-conditions. They are used to improve testing via runtime checking as well as enable static contract verification. In our approach, the contracts serve as a partial correctness specification of PAT. All test cases are then executed to make sure that code contract violation is absent. We highlight that code contracts are not only used to capture coding assumptions on data structures or model checking algorithms but also assumptions on GUI.

Like any model checker, PAT supports many options to apply model checking in different settings, for instance, whether to use Depth-First-Search (for memory saving) or Breadth-First-Search (for shortest counterexamples); whether to apply partial order reduction; whether to apply fairness consumption while verifying liveness properties; etc. The options may conflict with each other. For instance, fairness is irrelevant when the property is safety; partial order reduction is not sound when strong fairness is assumed. The options are carefully controlled through complicated logic on user interfaces, which can be specified by code contracts.

In order to achieve another level of assurance, we develop interface models (in PAT's input language) to capture all possible scenarios in which PAT or part of it is executed. For instance, for any complicated underlying data structure, we develop an interface model which subsumes all possible ways that PAT interacts it. Models can also be developed to capture all possible ways of users interacting with PAT through GUI. The models allow us to systematically generate test cases and, better, apply model checking techniques. PAT is firstly extended to support user defined C# library, i.e., an object or method defined the C# library can be invoked as part of a model. This creates a way of linking events of the interface models with actual codes of PAT. A transition of the model is thus the result of executing certain code fragments of PAT. For instance, the event of clicking certain button in the model for user behaviors generates an actual button clicking event. *The models are then model checked using PAT* - so that all possible sequences of executing PAT codes are verified against the embedded code contracts and, in addition, properties of entire system execution history. Notice that in order to model checking the interface model for a data structure, because a data structure may often take infinite different value, empirical studies are applied to discover reasonable bounds for the values. Interface models can be developed and verified for any part of PAT, which makes this approach compositional.

We remark that the interface models may contain concurrency, which makes model checking meaningful as well as challenging. Firstly, PAT supports parallel model checking [20], which makes use of multiple CPUs to explore different parts of a system concurrently. As a result, the underlying data structure may be accessed concurrently. Secondly, PAT supports multi-threaded graphic user interface and therefore multiple simulators and model checkers can be opened simultaneously, which leads to concurrent executions. By model checking the interface models, contract violations which are the result of unlikely event sequences can be discovered systematically. For instance, bugs on GUI which are the result of a particular sequence of button clicking on multiple PAT windows have been discovered. Even though this paper is focusing on model checking the model checker itself, the approach of combining code contracts with the model checker is much more general and this approach can be applicable for checking many C# software systems.

2 PAT Background

PAT (Process Analysis Toolkit) [27] is developed as a self-contained environment to support system modeling, simulation and verification. It has user friendly model editor, animated simulator as well as fully automated model checking facility. PAT offers

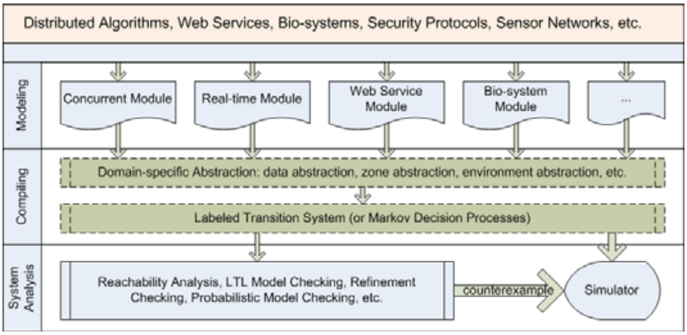

Distributed Algorithms, Web Services, Bio-systems, Security Protocols, Sensor Networks, etc.
Modeling
Concurrent Module
Real-time Module
Web Service Module
Bio-system Module
...
Compiling
Domain-specific Abstraction: data abstraction, zone abstraction, environment abstraction, etc.
Labeled Transition System (or Markov Decision Processes)
System Analysis
Reachability Analysis, LTL Model Checking, Refinement Checking, Probabilistic Model Checking, etc.
counterexample
Simulator

Fig. 1. PAT Architecture

a library of various model checking techniques for checking deadlock-freeness, linear temporal logics (with a variety of fairness), and refinement checking. Advanced state reduction techniques have been implemented, e.g., partial order reduction, process counter abstraction, parallel model checking. PAT has been used to model and verify a variety of systems. Previously unknown bugs have been discovered [19].

As shown in Fig. 1, PAT adopts a layered design to support analysis of different domains. For each supported domain (e.g., distributed systems, real-time systems, service oriented computing and so on), a dedicated module is created in PAT, which identifies the specialized language syntax, well-formness rules as well as formal operational semantics. The operational semantics translates a model into LTS (Labeled Transition Systems)[1] at runtime. LTS serves as a shared implicitly internal representation of the models, which can be automatically explored by the verification algorithms or used for simulation. To perform model checking on LTSs, the number of states in the LTSs must be finite. For systems with infinite states (e.g., with real time clocks or infinite number of processes), abstraction techniques are needed. Example abstraction techniques which have been realized include process counter abstraction, clock zone abstraction, environment abstraction, etc. Depending the property to verify, a proper verification algorithm is invoked. The algorithm performs on-the-fly exploration of an LTS. If a counterexample is identified during the exploration, it can be animated in the simulator. This design allows new modules to be easily plugged in and out, without recompiling the core system. This design achieves *extensible architecture* as well as *module encapsulation*.

Notice that the library of model checking algorithms as well as the GUI are shared by all modules. Their correctness are thus vital. The algorithms heavily reply on multiple highly complicated data structures, i.e., the one for system configuration; the one for compact representation of the system transition relation; the one for constraints on system clocks; etc. Not only the data structures must function correctly but also they must function efficiently. They are highly optimized, which implies that they may not be designed for rigorous system maintenance. For instance, having object orientation and recursion may not be feasible. The logic for controlling GUI is highly nontrivial as well. This is because the GUI controls different options for invoking the model checking algorithms. Different modules may employ different abstraction techniques which

[1] To be precise, it is a Markov Decision Process when probabilistic choices are involved.

conflict with certain group of model checking algorithms. For instance, over approximation of system graph conflicts with valid result for deadlock-freeness checking, i.e., an over approximated state graph is deadlock-free does not imply anything about the ordinary state graph. In such a case, if an abstraction which results in over approximation (e.g., predicate abstraction) is detected, verification result for deadlock-freeness checking must be modified properly.

PAT has been heavily tested with thousands of black-box test cases, and used daily by research and industry users. Nonetheless, bugs are still reported from time to time. Main reason for these bugs is that PAT is constantly under revision. As coding assumptions are often made in order to gain efficiency, code modification or function extension in one part of PAT often leads to coding assumption breaking in other parts, which results in new bugs. Currently, PAT has more than 600 KLOC, more than 1300 classes, 6 modules and more than 10K builds. PAT has attracted more than 800 registered users from more than 180 organizations. In summary, after years of development, PAT becomes a huge software package which requires systematically quality control.

3 Embedding Code Contracts

Code contracts [4] take the form of object invariants, preconditions, post-conditions and assertions. In a systematic way, it offers programming by design. Code contracts can be integrated into existing coding projects seamlessly, which makes them more attractive than approaches like SPEC#. Contracts are validated at run-time[2]. In the PAT project, coding assumptions are everywhere. One reason is that assumptions often make it possible to significantly simplify codes, which leads to faster model checking. Code contracts are used to capture coding assumptions on operational language semantics, model checking algorithms, underlying data structures, GUI, static model analysis functions, etc. In the following, we illustrate how code contracts are embedded systematically in PAT using two examples, one for a complex data structure which is essential to model checking real-time systems and the other for a user interface. Embedding code contracts is the first and the most essential step in our approach. They serve partially as a correctness specification of PAT.

Example 1 (Contracts for the DBM Class). Practical systems which interacts with the physical environment are often subject to quantitative timing constraints. For instance, a pacemaker must react to an abnormal heart condition within a critical time frame. Model checking real-time systems often involves manipulating constraints on multiple real-valued clocks. A timing constraint in PAT is the conjunction of multiple simple constraints. A simple constraint is of the form $c \sim d$ where c is a clock; d is a rational number; $\sim$ is a binary operator like $\geq$, $\leq$, etc. Multiple simple constraints may conflict with each other and thus make their conjunction unsatisfiable. For instance, the conjunction of $c_1 \geq 5$ and $c_2 \leq 1.5$ is unsatisfiable if c_2 is started within 3 seconds after c_1 is started (so that $c_1 - c_2 \geq 3$). During system exploration, the constraints must be stored, updated and solved efficiently. In PAT, this is achieved by techniques based on Difference Bound Matrix (DBM [9]).

[2] Contracts supports static analysis as well, which is helpful but largely irrelevant to this work.

```
public sealed class DBM {
1.      private List⟨List⟨int⟩⟩ Matrix;                             //the matrix itself
2.      private bool IsCanonicalForm = true;                        //a boolean flag
3.      private List⟨int⟩ Clocks;                                   //a list of clocks
4.      //Contract Invariant Method
5.      protected void ObjectInvariant() {
6.              Contract.Invariant(!IsCanonicalForm || (IsCanonicalForm && isCF()));
7.      }
8.      //methods
9.      public void AddClock(byte cID) { ···}                        //add a new clock
10.     public void ResetClock(byte cID){ ···}                       //reset an existing clock
11.     private void GetCanonicalForm() { ···}                       //Floyd-Warshall algorithm
12.     public void AddConstraint(byte cID, OperationType op, int constant){ ···}
13.     public void Delay(){ ···}                                    //let arbitrary time pass
14.     public bool IsSatisfiable(){ ···}                            //check satisfiability
15.     public DBM RemoveClocks(List⟨byte⟩ activeClocks) {           //remove clocks
16.             Contracts.Requires(IsCanonicalForm, "precon, failed.");
17.             Contracts.Requires(Clocks.containsAll(activeClocks), "precon. failed.");
18.             Contracts.Ensures(Matrix.Count == Clocks.Count && ··· &&
19.                                     IsCanonicalForm, "postcondition failed.");
20.             ···
21.     }
22.     ···
23.}
```

Fig. 2. DBM Contracts

Given n clocks $c_1, c_2, \cdots, c_n$, a DBM contains $n+1$ rows, each of which contains $n+1$ elements. Let d^i_j represent entry at i-th row and j-th column in the matrix. d^i_j represents the difference between clock c_i and c_j. A DBM represents the following constraint: $\forall\, i : 0 \mathinner{..} n.\ \forall\, j : 0 \mathinner{..} n.\ c_i - c_j \leq d^i_j$ where c_0 is set to be 0 all the time. The most important property of DBM is that there is a relatively efficient procedure to compute the tightest bound on each clock difference, which can be used to tell whether the constraint represented by the DBM is satisfiable or not. If the clocks are viewed as vertices in a weighted graph and the clock difference as the label on the edge connecting two clocks, the tightest clock difference is the shortest path between the respective vertices. The Floyd-Warshall algorithm [11] thus can be used to compute the tightest clock differences. A DBM which contains only tightest bounds is said to be in its *canonical form*. Given a DBM in canonical form, checking whether the constraint is satisfiable or not is as easy as checking if entry d^0_0 is positive.

DBM is implemented as a stand alone class of 1.5 KLOC. It makes use of some other simple data structures. Fig. 2 shows partially the signature of the DBM class, i.e., public methods and three relevant variables. *Matrix* is a two dimensional array storing the matrix itself; *IsCanonicalForm* is boolean flag to indicate whether the matrix is in its canonical form; *Clocks* maintains a list of active clocks. Sample code contracts are presented and underlined in Fig. 2. The invariant (line 5 to 7) states that either the DBM is not in its canonical form or if it is, then applying an alternative method for calculating the tightest bounds (which is implemented as method $isCF()$) makes no change.

Many of the methods require that the DBM must be in its canonical form before their execution. One example is $RemoveClocks$ which removes in-active clocks and together with constraints on them. If the DBM is not in its canonical form, removing clocks might weaken the constraint. For instance, if the constraint is $c_1 \geq 3$ and $c_2 \leq 6$ and $c_1 - c_2 \leq 1$ (which implies $c_1 \geq 5$), removing c_2 results in a weaker constraint $c_1 \geq 3$. Fig. 2 shows how the pre-condition (line 16 and 17) as well as post-condition is coded as contracts (line 18 and 19). The precondition states that the DBM must be in its canonical form and the clocks to remove must be present, while the simplified postcondition states that only the clocks in $activeClocks$ remain and the DBM remains in its canonical form. Notice that the postcondition is incomplete.

Efficiency is essential to any model checker. Keeping a DBM always in its canonical form (by calling method $GetCanonicalForm$ every time the matrix is modified by $AddClock$, $AddConstraint$, etc.) is infeasible given that the Floyd-Warshall algorithm is cubic in the number of clocks. Preferably, method $GetCanonicalForm$ shall only be invoked when necessary, i.e., in method $IsSatisfiable$. The pre-condition of the methods thus must be ensured by invoking the methods in particular orders. The assumptions on orders of method invocation can be referred as *class interface contracts*. To the best of our knowledge, such contracts are not supported by the code contracts project. It is supported, recently by the SPEC explorer project for model-based testing [3]. In the next section, we show that we can build an interface model to capture *class interface contracts*, and then not only generate test cases from the model but also verify the models against meaningful properties. □

Example 2 (Contracts for GUI). There are dozens of windows that users can interact with PAT, e.g., a featured editor which has many advanced editing functions; a simulator which allows user to perform different simulation functions; and a model checking window which controls all options for applying model checking. Given that PAT supports a library of model checking algorithms as well as optimization techniques, there could be a large combinations of options to choose from when a specific model checking problem is presented. For instance, whether it should be LTL model checking or refinement checking; or whether to apply nested DFS or SCC-based search for LTL model checking; or whether the LTL model checking should be based on generating Büchi automata. There are more than 5 options for generating Büchi automata from LTL alone [12]! The options are all controlled by enabling/disabling GUI components, which as a result has a complicated and error-prone control logic. The constraints are naturally captured using object invariants associated with the GUI components.

Fig. 3 illustrates the idea with one invariant associated with one checkbox in the model checking window. The checkbox, once checked, requires the model checking algorithm to produce one shortest witness trace (often as a counterexample). If the selected property is a safety property (e.g., a reachability condition, deadlock-freeness, refinement relationship), breadth-first-search based reachability analysis or refinement checking is applied. If the selected property is a liveness property, the checkbox must be unchecked and disabled. This is because a counterexample to a liveness property must be an infinite trace (which forms a loop in finite state systems). Instead, other GUI

```
protected void ObjectInvariant() {
        Contract.Invariant(checkBox_ShortestTrace_Invariant());
        ···;
}
private bool checkBox_ShortestTrace_Invariant() {
        if (Label_SelectedAssertion ! = null)
                if((AssertionType)cb_item.Tag == AssertionType.Liveness) {
                        return !CheckBox_ShortestTrace.Checked
                                                && !CheckBox_ShortestTrace.Enabled;
                } else { return CheckBox_ShortestTrace.Enabled; }
        return true;
}
```

Fig. 3. GUI Contracts

components like a dropdown list for fairness options, which only makes sense with liveness properties, must be enabled for selection.

In this example, the invariant states that if an assertion has been selected (from a table, which triggers update of a label *Label_SelectedAssertion* to reflect user's choice), and the selected property is a liveness property, then the checkbox must be disabled. Notice that this invariant is relaxed during the process of GUI updating. □

The contracts serve partly as a specification of the program. Once the code contracts are embedded, we firstly re-run all the test cases checking for contract violations. Our experience suggests that it is indeed possible that a test case is successful (i.e., causes no exception and produces correct output) but triggers violation of contracts during execution. One of the reasons is that the pre-condition may be irrelevant to the correctness of the output in certain cases. Detecting such cases are nevertheless useful as it helps to either find bugs or refine the specification (e.g., weakening the contract for pre-condition).

Because PAT is frequently updated, relevant code contracts must be updated as well. Coding assumptions may often change when a system evolves. Coding them explicitly as part of the system allows us to quickly detect bugs which are due to changing coding assumptions (refer to an example in Section 5).

4 Model Checking PAT

While code contracts are good at capturing intra-class coding assumptions, they are not good at capturing class level or even inter-class coding assumptions. For instance, the DBM class in PAT is designed to function correctly only under the assumption that its methods shall only be invoked in certain orders. In general, making such assumptions may not be reasonable. It is, however, common to model checkers as the assumptions may often be useful for the sake of efficiency. To test classes with implicit assumptions, all meaningful test cases must obey the assumptions. Otherwise, contract violation or even exceptions that are irrelevant to the correctness of the system may be reported. An interface model thus can capture all class-level or inter-class coding assumptions.

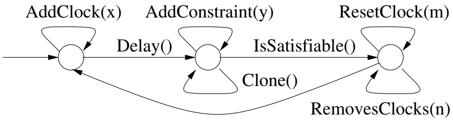


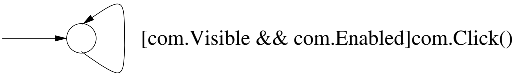
Fig. 4. DBM Model

Interface Modeling. In the following, we develop interface models to capture all valid ways of interacting with certain components of PAT or PAT as a whole. We illustrate the idea using two interface models.

Example 3 (Interface Model for DBM). Fig. 4 presents an interface model for the DBM class. The model takes the form of a finite state machine, possibly with auxiliary variables. The transitions are labeled with public methods of the DBM class. The model in Fig. 4 captures multiple class level assumptions.

For instance, method *RemoveClocks* is always be invoked after *IsSatisfiable* (and a method for collecting all active clocks defined in a separate class, which is omitted for the sake of space). Method *IsSatisfiable* invokes *GetCanonicalForm* internally and therefore it is unnecessary in method *RemoveClocks* to apply the Floyd-Warshall algorithm or even check whether variable *IsCanonocal* is true or not - it is always true. Similarly, this is also the case for *ResetClock*. Notice that this model is *open* to environmental inputs on the method parameters. □

Example 4 (Interface Model for GUI). The following presents an interface model for *any* user interface class. Let *com* be any GUI component (e.g.,any button), which users can interact with.



The model states that as long as *com* is visible and enabled, users can interact with it. Informally speaking, it means that no coding assumptions shall be placed upon the users and all user behaviors must be properly handled! Notice that this model can be generated automatically from the signature of any GUI class.

Users can interact with multiple user interfaces simultaneously. Therefore, the interface model for PAT contains the parallel composition of multiple interface models. Furthermore, the interface models may communicate with each other. □

Once we have the model, the task of verifying this part of the system breaks into two sub-tasks. Firstly, we need to guarantee that in the real system, interactions with the objects (of the class or classes) are permitted by the model. In this project, this is achieved with the expert knowledge of the PAT developers. In general, techniques like program slicing [29] may help, or a run-time monitor program, much like the monitor for code contracts, can be used to detect violation of class interface violation. Secondly, we need to guarantee that the system functions correctly given any behavior allowed by the model. One way of checking that is to systematically generate test cases from the models and execute the test cases while looking for exceptions or code contracts violation [3]. Alternatively, we can model checking the models! By model checking, code contracts which are related to the entire system execution history (e.g., liveness properties) can be stated as a property and then formally verified. The models can be captured as PAT models and, hence, *we can model check the models using PAT itself!*

Supporting Runtime C#. In order to verify actual source codes, firstly we extend PAT to support dynamic loading of C# code inside a model. Our approach is to compile a C# program into a dynamic-link library (DLL) and use it during system exploration via *reflection*. By following pre-defined API, any C# class can be invoked dynamically in an interface model in PAT. Furthermore, the C# class may contain code contracts. PAT provides a contracts compilation option to automatically compile the code using contracts rewriter compiler. Any contract violation or run-time exception is presented as a runtime error to the users during model checking. In other words, ordinary PAT codes, with coding assumptions, can now become part of a PAT model with little modification.

Checking PAT. The following approach is adopted to model check PAT. Firstly, an interface model of a PAT component is written as a PAT model; the relevant C# source codes, with code contracts embedded, are compiled into an external DLL and then PAT is used to enumerate all possible behaviors of the model. If any event sequence triggers a violation of code contracts or exceptions, then a possible bug is detected. Because it is the actual code which is being executed during model checking, a discovered bug corresponds to an actual bug.

Like testing, this approach is incremental - a self-contained class can be firstly modeled and checked and then classes which rely on it can be modeled and checked. If the states searching in the model checking is viewed as simply a systematic way of generating test cases, this approach is closely related to work on model-based testing. It is, however, more than testing. For model checking, meaningful properties, which are implied from the correctness of system and cannot be validated by testing, can be verified. In general, this approach is as challenging as verification software. Many challenges like infinite data value and asynchronous thread execution must be dealt with. In the following, we use the two examples to illustrate relevant issues and our remedies.

Example 5 (Model Checking the DBM Class). A model may be open to environmental inputs. For instance, given the model for DBM class presented in Fig. 4, a clock must be supplied when an event linked to method $AddClock$ is invoked; or a simple constraint must be supplied when invoking method $AddConstraint$. In order to model check the model, it must be closed by supplying an environment. In general, there are infinite possible clocks or constraints. This problem is solved by applying empirical studies to discover reasonable bounds for the values.

Given the DBM model, we need to fix a finite set of clocks (so that x, m, n in Fig. 4 have finite values) as well as a finite set of constants for forming clock constraints (so that y has finite values). It is discovered that for most of the real-world real-time systems verified by PAT, the number of clocks are often limited to a small number (e.g., 10 or less). This is not surprising as PAT is optimized to minimize the number of clocks [28] - recall that Floyd-Warshall algorithm is cubic in the number of clocks. PAT

Table 1. Experiments: Reachability/deadlock-freeness

#DBM	#Clocks	#Constants	Reachability/deadlock		Every clock is bounded	
			#States	Time (s)	#States	Time (s)
1	1	6	7375	1.55	63641	31.4
1	1	7	12947	3.12	127362	74.1
1	1	8	21235	5.88	234254	157
1	1	9	33007	10.3	403274	310
1	2	6	473328	168	5918993	5036
1	2	7	1104560	449	15783113	15831
1	3	3	222903	64.1	2016851	1264
1	3	4	1532935	572	17659797	15431
2	1	3	511225	172	?	?

only introduces a clock whenever necessary; a clock is shared as much as possible and a clock is removed as soon as possible. Furthermore, there are only a relatively small number of constants for forming clock constraints in most cases. Once we fix the clocks and constants, we have a finite model which is subject to model checking.

Because PAT supports parallel model checking, any data structure used by the model checking algorithms may be accessed in parallel by multiple threads. This may create problems like race condition. Even if the objects are not shared by the threads, static variables or references which are accessed by the objects directly or indirectly are always shared. Static variables allow quick access of information. They, however, must been properly locked and unlocked if accessed concurrently. We thus extend the model to capture concurrent accessing of data objects, e.g., the DBM, by composing multiple copies of the interface models in parallel. Furthermore, the exact locking mechanism used in PAT has been modeled and reflected in the model as well. Different properties can be formulated and model checked against the interface models. In general, a property is necessary condition of the correctness of the PAT. By verifying that the properties are true, we gain confidence in the system's correctness.

In the following, we illustrate three important properties of DBM and use PAT to model check the respective model. Firstly, an unsatisfiable reachability condition is model checked, which triggers exploration of the complete state space. This allows us to verify that the embedded code contracts are satisfied in all system configurations. In addition, we verify that the DBM model is deadlock-free. This is particularly interesting with more than one DBM objects - it should not be that two DBM objects are both waiting to access a shared object. The experiment results are summarized in column "Reachability/deadlock" of Table 1. The results are obtained on a PC with Intel Xeon 4-Core CPU*2, 32 GB memory, with fixed number of DBM objects, number of clocks and number of constants. This property is verified using an on-the-fly reachability analysis algorithm in PAT. After correcting serval bugs, the result is eventually all false, as expected. Secondly, a property stating that every clock is bounded is verified to confirm a fundamental theorem that every clock can take only finite different ranges (which is known as zones [27]). The theorem is one of two necessary conditions to guarantee that model checking of real-time systems in PAT is always terminating. It is an important property of DBM which can be proved from the formal semantics of the real-time

Table 2. Experiment C: Every clock always eventually expires

#DBM	#Clocks	#Constants	#States	Result	Fairness	Time (s)
1	2	3	58234	false	no fair	26.7
1	3	3	1445821	false	no fair	2229
1	1	3	1676	false	weak fair	0.575
1	2	3	12372	false	weak fair	26.7
1	3	3	1445821	false	weak fair	2223
1	1	3	1676	false	strong fair	0.643
1	2	3	58234	false	strong fair	29.7
1	3	3	1445821	false	strong fair	2348
1	1	3	5020	true	global fair	2.19
1	2	3	148860	true	global fair	353
1	3	3	3462673	true	global fair	244733

system modeling language supported in PAT [27]. The experiment results are summarized in column "Every clock is bounded" of Table 1, where a question mark means out of memory. Lastly, Table 2 summarizes our experiments on verifying the other necessary condition, which too can be proved from the semantics model. Intuitively, the property states that every clock *always eventually* expires. We highlight this is a liveness property which cannot be validated by testing. It is captured as an LTL formula and verified using the automata-based verification algorithm in PAT. Furthermore, this property is valid only under certain fairness constraint [27], which intuitively says that there are only finitely many system actions within one time unit. Notice that verifying LTL properties with strong fairness is a unique feature of PAT.

If a counterexample is generated, a unit test case is generated from the counterexample straightforwardly (since the counterexample is a sequence of method calls with input values) so as to locate the bug. As expected, the experiments show that model checking suffers from the state space explosion problem, e.g., parallel execution of multiple DBM objects results in huge number of system configurations. Furthermore, because of the bounds, only part of all possible behaviors are examined. Nonetheless, large number of system executions are examined systematically (with blind cases) against complex properties, which greatly increase our confidence in system correctness. Furthermore, because of the empirical studies, we focus on common cases and therefore reduce the probability of producing wrong results in common practice of PAT. □

Example 6 (Model Checking the GUI). In order to model check PAT as a whole, we build a model to capture all possible ways of users interacting with PAT. The model is composed of each and every user interface model in the order which users can interact with them. Each event in the model is linked to an actual GUI event which triggers execution of PAT. Model checking is then applied to enumerate all event sequences and validate them against the embedded contracts, including those which are hard to create in practice. For instance, the following scenario is found to lead to contract violation. A user firstly clicks one button to initiate a thread for state graph generation in one simulation window, which triggers an attempt to lock shared (static) variables. The user then clicks the same button in another simulation window. If two models happen to share common process definitions and two models are not identical, then the first simulation window may behave wrongly because an indirectly accessed variable is changed when the user clicks the second button.

The GUI model essentially creates a “robot” which controls PAT, which allows users to manipulate PAT arbitrarily. We can then easily create and verify complicated properties which may require multiple execution of multiple model checking algorithms. For instance, one interesting theory is that if a model satisfies a property under weak fairness, then it must satisfy the property under stronger fairness constraints like strong fairness or strong global fairness [27]; if a counterexample is produced when a property is verified under strong fairness, then a counterexample must be produced when the property is verified under weak fairness. In order to check this is indeed true, a simple GUI model is created to load one built-in case study at a time, perform model checking under weak fairness, perform model checking under strong fairness and then compare whether the results are as expected. Many theorems implied from the correctness of PAT can be checked in this way.

One difficulty in checking GUI models is that different GUI components may run as different threads. Multiple threads may execute simultaneously. For instance, one occurrence of the event for clicking the verification button triggers the creation of a thread for model checking. Before the model checking completes, the user can click another button to trigger another thread. The threads are scheduled by the system scheduler. Different executing of the same event sequence of the model may result in different system configuration due to different run-time scheduling. This is a known problem to GUI testing or testing of concurrent programs in general. A common remedy is to run a test case sufficient number of times (or with inserted random thread sleep) so as to exhaust all possible scheduling. Notice that because the model checking of the GUI model largely depends on the size of the input model, we omit the statistics.

5 Discussion

In this section, we discuss the limitation of our approach and related works.

Limitations. Firstly, *we cannot completely verify PAT.* Compromises have been made in order to deliver a useful technique handling PAT. For instance, the code contracts only capture part of the correctness specification; we can only verify part of the behaviors of an interface model, etc. Secondly, when model checking a component of the system, assumptions on the rest of the system are often necessary. For instance, the input to method $RemoveClocks$ is assumed to be a set of clocks, which assumes that the method for obtaining the clocks removes any redundancy. Systematically verifying the assumptions are highly non-trivial. This is a known problem which has been discussed in [6]. Lastly, generating properties to be model checked needs expert knowledge on the underlying theories of the system.

The infamous state-space explosion problem still exists. For instance, given a DBM object with N clocks, there are 2^N different inputs to method $RemoveClocks$. This problem is known to be best solved by methods like data abstraction [2], which currently remains as one of our future work. But still, model checking remains useful even if only part of the system behaviors are explored. It explores all possible behaviors of a model, including corner cases which are unlikely for real-world applications. Compared to model-based testing, the additional properties are often useful in gaining confidence of the implementation or the theorems which lead to the properties. Our experience is that given all model checking algorithms have multiple theorems behind (e.g., for soundness, completeness and termination), many properties can be deduced naturally.

Related Work. To our best of knowledge, this work is the first attempt on using advanced system analysis techniques (e.g., code contracts and model checking) to systematically validate a model checker. Our approach is related to numerous work on software verification, testing and debugging.

Our approach relies on programming by contracts. In particular, the code contracts project essentially makes it possible [4]. There are other methods for embedding specification into programs. Noticeable examples are SPEC# for C# and JML for Java. We remark that as long as there is a way of run-time checking the specification, any method would work in our approach. We choose code project simply because it is the option which requires minimum modification to our programs.

Our approach can be categorized as combining programming by contracts with model checking. It is closed related to work on combining programming by contracts with model-based testing. The tool SPEC explorer supports model-based testing in addition to contract embedding [3]. Similar to our approach, SPEC explorer uses a model checker so it can enumerate all possible sequences of method invocations that do not violate precondition or invariant of the system's contracts. Furthermore, users are allowed to specify a set of testing properties, which plays a similar role as our interface model. Our approach is different from model-based testing [16] in the following ways. Firstly, we perform model checking instead of model-based testing. The difference is that besides exception-freeness and no violation of contracts, additional properties regarding to interface models can be verified. The properties may not be validated by testing. The properties are often implied from the underlying theorems. By model checking them, not only we gain confidence on PAT but also the theorems. In addition, our work targets at validating a model checker. We use the model checker to check the interface models as a way to verify itself.

This work is related to work on software verification [1,26]. Apart from very constrained scenarios (e.g., verification of device drivers), the software verification tools are not widely used in general software development process. The main reason is that they do not scale. For instance, the SLAM project is based on data abstraction, which is a complicated and computationally expensive process. In contrast, our approach is scalable. Even partial code contracts are useful. Like testing, our approach is compositional in the sense that each time part of PAT can be checked and then their composition. Furthermore, it is compatible with rapidly evolving programs. Once a few relevant contracts need to be updated every time system evolves. As a reasonable price to pay, we can not verify PAT altogether. Nonetheless, our approach helps to significantly improve stability and reliability.

The work is related to work on specification and verification of object interfaces [15,5]. The main difference is that we combine code contracts. This work is related to work on combining testing with model checking [13,18,23]. In [18], a tool named

UnitCheck is present which allows creation, execution and evaluation of testing cases using the Java Pathfinder model checker. Unlike [18] where models are automatically extracted from programs, interface models are provided by PAT developers as an additional code contracts. In addition, this work is remotely related to work on testing of concurrent software [24,25], GUI testing and testing of evolving programs.

6 Conclusion

Model checkers are specialized software whose correctness are vital. In this work, we propose the combination of code contracts and model checking as a way to systematically improve their quality. The combination is effective in combating the complexity of software. Three levels of system specification are handled using the proposed approach. First is the specification of a method or a single class, captured in the form of pre/post-condition or class invariants and validated using run-time checking facility from the code contract project. Second is safety properties of a single class or a group of coupled classes, captured using interface models. It can be verified by model-based testing or reachability analysis. Lastly, specification of the entire system execution are verified against the models by model checking techniques.

We experiment the approach in checking the correctness of the PAT model checker. Through the experiments, we discovered multiple bugs, and gained more confidence of PAT. One of our future is to combine data abstraction (e.g., predicate abstraction) in model checking the interface models.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. STTT 5(1), 49–58 (2003)
3. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The SPEC# Programming System: Challenges and Directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
4. Barnett, M., Fähndrich, M., de Halleux, P., Logozzo, F., Tillmann, N.: Exploiting the Synergy between Automated-test-generation and Programming-by-contract. In: ICSE Companion 2009, pp. 401–402. IEEE, Los Alamitos (2009)
5. Bierhoff, K., Aldrich, J.: Lightweight Object Specification with Yypestates. In: ESEC/SIGSOFT FSE 2005, pp. 217–226. ACM, New York (2005)
6. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Jurdzinski, M., Mang, F.Y.C.: Interface Compatibility Checking for Software Modules. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 428–441. Springer, Heidelberg (2002)
7. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
8. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)

9. Dill, D.L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
10. Emerson, E.A., Wahl, T.: Dynamic Symmetry Reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
11. Floyd, R.W.: Algorithm 97: Shortest Path. Commun. ACM 5(6), 345 (1962)
12. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
13. Gunter, E.L., Peled, D.: Model checking, Testing and Verification Working Together. Formal Asp. Comput. 17(2), 201–221 (2005)
14. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
15. Hughes, G., Bultan, T.: Interface Grammars for Modular Software Model Checking. In: ISSTA 2007, pp. 39–49. ACM, New York (2007)
16. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
17. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)
18. Kebrt, M., Sery, O.: UnitCheck: Unit Testing and Model Checking Combined. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 97–103. Springer, Heidelberg (2009)
19. Liu, Y., Pang, J., Sun, J., Zhao, J.: Verification of Population Ring Protocols in PAT. In: TASE 2009, pp. 81–89. IEEE Computer Society, Los Alamitos (2009)
20. Liu, Y., Sun, J., Dong, J.S.: Scalable Multi-core Model Checking Fairness Enhanced Systems. In: ICFEM 2009. LNCS, vol. 5885, pp. 426–445. Springer, Heidelberg (2009)
21. Mühlberg, J.T., Lüttgen, G.: Blasting Linux Code. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
22. Peled, D.: Combining Partial Order Reductions with On-the-fly Model-Checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
23. Peled, D.: Model Checking and Testing Combined. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 47–63. Springer, Heidelberg (2003)
24. Sen, K., Marinov, D., Agha, G.: CUTE: a Concolic Unit Testing Engine for C. In: ESEC/SIGSOFT FSE 2005, pp. 263–272. ACM, New York (2005)
25. Sherman, E., Dwyer, M.B., Elbaum, S.G.: Saturation-based Testing of Concurrent Programs. In: ESEC/SIGSOFT FSE 2009, pp. 53–62. ACM, New York (2009)
26. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. ACM Trans. Softw. Eng. Methodol. 17(2) (2008)
27. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
28. Sun, J., Liu, Y., Dong, J.S., Zhang, X.: Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In: ICFEM 2009. LNCS, vol. 5885, pp. 581–600. Springer, Heidelberg (2009)
29. Weiser, M.: Program Slicing. In: ICSE, pp. 439–449 (1981)