PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers

Yang Liu
National University of Singapore
liuyang@comp.nus.edu.sg

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

Jin Song Dong
National University of Singapore
dongjs@comp.nus.edu.sg

ABSTRACT

Model checking is emerging as an effective software verification method. Although it is desirable to have a dedicated model checker for each application domain, implementing one is rather challenging. In this work, we develop an extensible and integrated architecture in PAT3 (PAT version 3.*) to support the development of model checkers for wide range application domains. PAT3 adopts a layered design with an intermediate representation layer (IRL), which separates modeling languages from model checking algorithms so that the algorithms can be shared by different languages. IRL contains several common semantic models to support wide application domains, and builds both explicit model checking and symbolic model checking under one roof. PAT3 architecture provides extensibility in many possible aspects: modeling languages, model checking algorithms, reduction techniques and even IRLs. Various model checkers have been developed under this new architecture in recent months. This paper discusses the structure and extensibility of this new architecture.

1. INTRODUCTION

Software development has entered a mass production era. To ensure quality, software verification is becoming a compulsory step in the software development life cycle, especially for safety and critical systems. Among the principal validation/verification methods (e.g., simulation, testing and theorem proving), model checking [12] has emerged as a promising and powerful approach to automatically verify software systems. Model checking has been successfully used in practice to verify complex circuit design, communication protocols, driver software, software process models, software requirement models, architectural frameworks, design models, product lines and system implementations.

Model checking is the application of an automatic process to formally verify whether a *model* satisfies a *property* by exhaustively exploring the *state space* of the model. Till now, model checking has become a wide area including many different approaches (e.g., explicit model checking and symbolic model checking) catering for different properties (e.g., temporal logics, refinement relationship, etc.) and state space reduction techniques (e.g., partial order reduction, symmetry reduction, etc.). Applying model checking in a new application domain requires in-depth understanding of model checking techniques. Unfortunately, the complexity prevents many domain experts, who may not be experts in the area of model checking, from successfully applying model checking to their domains. In the following, we examine the challenges of adopting model checking techniques.

Using existing model checkers
The most commonly used approach is to use existing model checkers (e.g. NuSMV [10], SPIN [18] and FDR [27]). The difficulty of this approach is explained below. First, the learning curve is steep. To precisely understand the semantics of the language and various verification options is highly non-trivial. Second, existing model checkers may be inefficient or insufficient to model domain specific applications, due to lack of language features or data structures. For example, multi-party barrier synchronization is difficult to achieve in SPIN model checker. Translating a domain-specific model and properties into the input language of a general purpose model checker may often be ad hoc.

Extending existing model checkers
To support a new application domain, users often need to customize the existing model checkers for language extension, different state encoding, new verification algorithms or reduction techniques. Existing tools are generally complicated and highly coupled in order to support efficient verification. For example, SPIN generates a verifier program in C by combining code fragments (in string format) and user input model rather than direct verification. The source code of PRISM [16] is mixed with Java and C++. To understand the source code is generally difficult or infeasible.

Developing a new model checker Developing a model checker from scratch is even more challenging. The basic functionalities of a model checker include language parsing, system simulation, verification algorithms, state reduction techniques and counterexample generation and display. To finish all with a sound system requires years of efforts. All established model checkers (e.g., SPIN, NuSMV, UPPAAL [4] and PRISM) have 10 to 20 years history.

To tackle the above challenges, we propose an extensible

framework called PAT3 (Process Analysis Toolkit version 3) [1], which facilitates effective incorporation of domain knowledge with model checking techniques. Starting from a simple model checker, PAT1 (version 1) [31] supported the model and verification of concurrent state-rich systems with various fairness conditions. PAT2 (version 2) [23] focused on reusing the core model checking algorithms in the concurrent module of PAT1, e.g. web service reasoning module. PAT3 (this work) does a complete architecture redesign that aims to support systematic construction of verification systems for wider application domains and extensions in various aspects. The contribution of this work is three-fold.

Firstly, we propose a novel framework with layered architecture (see Fig. 1 in page 3). Previous development on model checkers focuses on effectiveness with a specific modeling language, e.g., SPIN, NuSMV, FDR, PRISM, etc. Extensibility is never a concern in these tools. Later, the need for algorithm reuse and extensibility has been addressed by frameworks like Bogor [14], Model-Checking Kit [28], JPF [32], etc. These tools have the similar design as PAT2, hence extensions and customizations are limited (see Section 7 for details). Compared with these frameworks, PAT3's architecture design has the following advantages. (1) The layered design reduces the coupling between different components. Particularly, the intermediate layer separates modeling languages from verification algorithms completely. This design facilitates the reuse of existing model checking techniques and easy extension of modeling languages. PAT supports three intermediate representations (i.e. Labeled Transition System, Timed Transition System and Markov Decision Process), which allow PAT3 to support a wide range of modeling languages. To the best of our knowledge, PAT3 is the only model checker supporting verification for a hierarchical modeling language for concurrent, real-time (with dense-time semantics) and probabilistic systems[1]. (2) PAT3 has a modular design. Each application domain (or modeling language) is encapsulated into a plug-in module, which is loaded dynamically at runtime. Currently, 11 modules for different application domains have been developed, which demonstrates the practicality and scalability of our framework. (3) PAT3 integrates both explicit-state model checking and symbolic model checking (based on BDD [8] or SAT solvers) under one roof, which allows users to leverage the advantages of different verification approaches. Especially, we develop a BDD library that is embedded in PAT3 for encoding compositional operators (e.g., interleave and choices operators). With it, users can generate BDD encoding for concurrent systems without knowing the Boolean formulae.

Secondly, PAT3 framework is specially designed for extensibility, which provides the different interfaces (APIs) for domain experts to create customized model checkers with minimum efforts. These interfaces allow PAT3 to be extended in the following ways.

- Translate external models into existing languages in PAT3 through module APIs.
- Extend the existing modeling languages with new syntax, data types or libraries.
- Extend PAT3 with domain specific model checking algorithms, state reduction techniques or abstraction techniques.
- Create completely new module in PAT3 to support a new language with the help of module generator tool.

Lastly, our engineering efforts make PAT3 a self-contained environment to support composing, simulating and reasoning of system models. It comes with cross-platform support, user friendly GUI, featured model editors and an animated simulator. More importantly, PAT3 implements a rich library of model checking techniques[2] catering for checking a variety of properties, e.g., deadlock-freeness, divergence-freeness, reachability, linear temporal logic (LTL) (with different fairness assumptions), refinement checking, real-time verification and probabilistic model checking, etc. Although PAT3 is designed for flexibility, this flexibility does not compromise the performance. Advanced optimization techniques are implemented in PAT3, e.g., partial order reduction, process counter abstraction, bounded model checking, and parallel model checking. We have used PAT3 to model and verify a variety of systems, ranging from recently proposed distributed algorithms, security protocols to real-world systems like the pacemaker system. Previously unknown bugs have been discovered. The experiment results (see Section 6) show that PAT3 is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in many cases.

The remainder of this paper is organized as follows. Section 2 briefly introduces model checking techniques. Section 3 presents the conceptual architecture of PAT3. Section 4 describes PAT3's design and implementation details. Section 5 explains all the possible ways to extend PAT3. Section 6 presents experimental results to demonstrate the effectiveness of PAT3. Section 7 reviews related work. Section 8 concludes the work with future works.

2. PRELIMINARY

Model checking [12] has been conceived as an automatic verification technique for finite state systems. It performs an exhaustive search of the state space of a model to determine if some property is true or not. If the result is negative, the user is often provided with a counterexample. The most serious disadvantage of model checking is the state explosion problem. The size of the global state graph can be (at least) exponential in the size of the program text.

2.1 Model Specification

System models can be specified in numerous different ways. In theory any means of formally specifying a model could be used for the purpose of model checking. For different modeling languages, though the syntax may vary dramatically, as long as they have the same *semantic model*, they can be verified using the same set of model checking algorithms. This is the foundation of our approach, i.e., to compile different modeling languages into a set of common semantic models. Three commonly used semantic models for concurrent system, real-time system and probabilistic system are explained below.

LTS is arguably one of the most general semantic models for un-timed systems. For instance, it is the semantic model of FSM, Process Algebra, Graph Transition System and so on. A timed system often has a timed transition system as

[1]Probabilistic Timed Automata have a flat structure.

[2]Techniques developed in PAT3 can be found in [1].

its semantic model. TTS is the common semantic model of Timed Automata and timed process algebras. If probabilistic system behaviors are relevant, a probabilistic semantic model is necessary. In this paper, we focus on Markov decision processes (MDP), which is expressive enough to capture probabilistic choices as well as concurrency.

2.2 Property Specification

System properties can be specified in different forms. A property can be specified in a dedicated language like temporal logics or in the same language which is used to specify the system model. The former is verified by dedicated temporal-logic based model checking, whereas the latter is verified by refinement checking. In the following, we show a 'semantic' classification, i.e., properties are divided into two categories: safety properties and liveness properties [3, 29].

Safety

Informally speaking, safety properties stipulate that "bad things" do not happen during system execution. A finite execution is sufficient evidence to the violation of a safety property. Hence, safety properties are often verified based on finite execution/trace semantics. Common safety properties include deadlock-freeness, reachability, invariance, or properties expressed in the form of finite state automata. Common refinement relationships like trace-refinement can also be categorized as safety properties.

Liveness

There have been different definitions of liveness [3, 29]. In this paper, we define liveness properties as those which are not safety properties. Informally speaking, liveness properties stipulate that "good things" do happen eventually. A counterexample to a liveness property is often[3] an infinite system execution, which forms a loop if the system has finitely many states. Therefore, liveness properties are often verified using loop-searching algorithms. Liveness properties are often specified using temporal logics, e.g. *Computation Tree Logic (CTL)*, LTL and *CTL**. In theory, they can also be specified by finite state automata with fairness condition, e.g., Büchi automata, Streett automata, etc.

2.3 Model Checking Approaches

Different models or properties often lead to different dedicated model checking approaches, in the name of efficiency. Distinguished by how states are stored and manipulated, there are two main paradigms for model checking: explicit state model checking and symbolic model checking.

The first proposal of model checking relies on exhaustive search through explicit representations of reachable system states [12], which is known as *explicit model checking*. This approach is adopted by SPIN. However, it suffers from the *state space explosion* problem due to the exponential increasing of the explicit states.

Symbolic model checking is introduced later to solve this problem by enumerating states symbolically (typically based on the notion of Binary Decision Diagrams (BDDs) [8, 9]), which avoids building the state space directly. This approach is adopted by the tool NuSMV. It has been shown that indeed BDDs can serve as an efficient representation of finite state machines and allow symbolic model checking to handle rather complex systems [5]. However, the worst case boundaries of the size of BDDs (which depends on the variable ordering) are catastrophic [8].

[3]Not always with our definition of liveness.

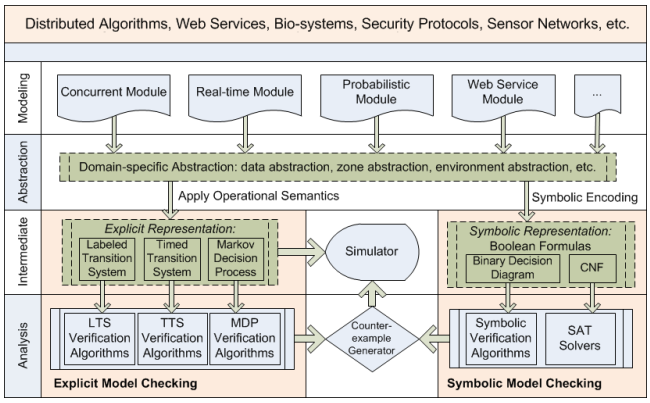

Figure 1: PAT3 System Architecture

Bounded model checking [11] has been proposed as an alternative symbolic model checking approach. The idea is to encode finite state machines (and the property to be verified) as a Boolean formula. The property is satisfiable if and only if the underlying state machine can realize a finite sequence of transitions that reaches states of interest, and then SAT solvers [2] can be used to produce counterexamples efficiently. If such a path segment cannot be found at a given length k, the search is continued for larger k.

In addition, many state reduction techniques have been proposed and realized in many existing model checkers, such as partial order reduction, symmetry reduction, predicate abstraction, etc. One of the drawbacks of symbolic model checking is that reduction techniques developed for explicit model checking cannot be directly applied.

3. CONCEPTUAL ARCHITECTURE

During the development of PAT1 and PAT2, extensibility becomes critical as we want to quickly apply existing model checking techniques to other application domains or implement new verification techniques. Code reusing can both speedup the development and reduce maintenance efforts. To achieve this goal, we need a common interface to separate the verification algorithms and domain specific modeling languages. Naturally, a prototype of a 3-layered design was adopted. To further articulate the architecture, we refined the design to 4 layers according to the three steps of model checking process (i.e., compilation, abstraction and verification). To integrate symbolic model checking into the system, a separate interface and related verification supports need to be developed. With several attempts, we arrived at an architecture design as shown in Figure 1. This goal of this design is to reduce the coupling between different components (hence increases the reusability of lower layers and the extensibility of upper layers) and integrate explicit and symbolic model checking under one roof.

3.1 Modeling Layer

The top level of the architecture is the supported application domains (e.g., distributed system, service oriented computing, security protocols and so on). For each application domain, modeling layer identifies the domain specific language syntax, well-formedness rules as well as formal

operational semantics, which are all encapsulated in a separate *module*. The main components in modeling layer are the language parser and model components (including syntax classes, variables, channels, etc.). Given an input model (in textual or graphic format), a parser checks the correctness of the syntax and generates model components. The (operational) semantics of the language shall be implemented in the syntax classes of the language constructs, which can be either explicit state representation or symbolic representation using Boolean formulae (details are explained in Section 4.2). For better encapsulation, all domain specific functions and components should be included in the modeling layer.

3.2 Abstraction Layer

Model checking techniques generally only work with finite state systems. When a system has infinitely many states according to its *concrete* operational semantics, (automated) state abstraction is essential. For example, a real-time system always has infinitely many states since there is an arbitrary small interval between any two time points. Hence abstraction techniques like zone abstraction [22] can be used to generate finite systems. Even when the state space is finite, effective abstraction/reduction techniques may reduce the state space significantly. For example, process counter abstraction [26] can group identical processes using process counter variables and ignore process identifiers (if they are irrelevant). In PAT3, the abstraction layer implements abstraction techniques as independent functions, which map a concrete state to an abstract state. These functions are invoked during the state space exploration to generate abstract states. Abstraction/reduction techniques, like partial order reduction, are language or algorithm dependent, which are then treated differently (see Section 5.5 for details).

3.3 Intermediate Representation Layer (IRL)

IRL contains different semantic models supported in PAT3. Each semantic model defines a state interface class with methods to drive the state space exploration. After compilation, the input model is converted to an initial state interface class. Then the state space can be generated on-the-fly starting from the initial state by following the operational semantics (and applying abstraction/reduction techniques). PAT3 supports all three semantic models mentioned in Section 2.1.

For explicit model checking, the state interface class has a number of operations, which allow the underlying model checking algorithms to drive the execution of the system and collect information from system states. The most important operation is the *MoveOneStep* method, which returns a set of transitions from the current system state. By repeatedly invoking this method, the model checking algorithms explore the whole state space. In model checkers like Kit [28], the complete semantic model (i.e., Petri nets in Kit) needs to be constructed first before the verification. However in PAT3, the verification algorithms use *MoveOneStep* method to explore the state space *on-the-fly*, i.e., only partial state space needs to be explored if a counterexample exists. Note that *MoveOneStep* method is implemented at modeling layer so as to capture the operational semantics of a language. Another important operation is the *GetID* method, which returns a unique (compact) identifier for any system state. The idea is that the identifier abstracts over the system state information (e.g., variable values, channel buffers, program counter and so on) using primitive values (e.g., int or string). The state interface can also be used by the simulator to show the system state space graphically.

For symbolic model checking, different operations are defined so that a symbolic representation is generated to capture the language semantics, usually in the form of Boolean formulae. For instance, for BDD-based symbolic model checking, a method *EncodeProcess* is defined for each language construct. The Boolean formulae are usually stored in the form of Binary Decision Diagram (BDD) for symbolic model checking or Conjunctive Normal Form (CNF) for bounded model checking.

3.4 Analysis Layer

This layer mainly contains reusable model checking algorithms. In the explicit model checking approach, a set of verification algorithms have been developed for each semantic model in IRL. For example, deadlock checking, reachability checking, LTL verification with fairness assumptions, refinement checking have been developed for LTS. The verification algorithms only invoke state interface to explore the state space. Therefore, the modeling language is separated from the verification algorithms completely. If the verification result is false, a counterexample is produced, which can be visualized via the simulator. For the symbolic model checking approach, symbolic verification algorithms are developed for the generated BDD encoding of the system. Alternatively, SAT solvers can be used for solving CNF equations for bounded model checking.

Remark.

This layered framework separates the model checking process into independent steps, which reduces the coupling significantly. In additional, other analysis techniques than model checking can also be integrated in analysis layer. For example, model-based testing can be applied based on the state interfaces, especially the internal representations for explicit model checking (i.e., LTS, TTS, MDP).

4. DESIGN AND IMPLEMENTATION

Model checking involves functions like state-space exploration, process scheduling, and state management. In the actual implementation, these functions are highly tangled to achieve optimal performance. Inspired by Bogor project [14][4], we apply Design Patterns [15] to encapsulate these functions. Hence, the coupling between the components is reduced and then extensibility can be achieved. Different from Bogor, we further apply this design strategy to module interface design. The dependency between common library and individual modules is minimized. We choose C# as the implementation language for the benefits of Object-Oriented design and competitive performance.

PAT3's design is hierarchical, as reflected in the class diagram in Figure 2. The system consists of two basic packages, namely *PAT.GUI* and *PAT.Common*, and 11 module packages. Each of the 11 modules implements necessary module interfaces and is packed into a plug-in DLL. The complete API can be found in both PAT tool and website [1].

[4]Bogor system design covers partial components in *PAT.Commom* project. See Section 7 for detailed discussion.

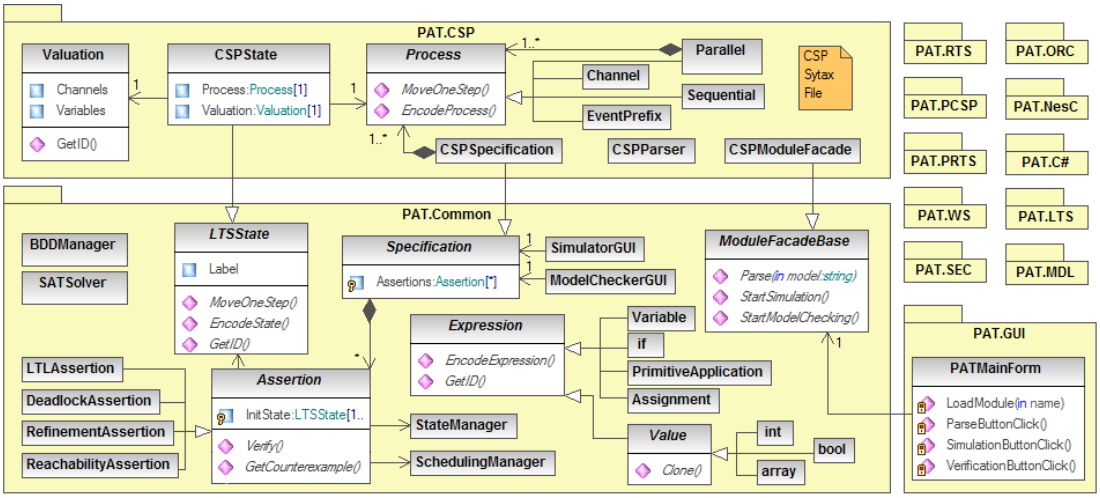

Figure 2: Class Diagram of PAT3

Note that abstract classes are preferred in PAT than interface mainly due to the performance reason in C#. We use the CSP module as a demonstrating example to explain the implementation. The modeling language in CSP module is called CSP# [30], which is an event-based modeling language for concurrent systems. CSP# is an extension of CSP [17] with shared variables and low-level programming constructs (e.g., assignment, if-then-else, while-loop, etc). The semantical model of CSP# is LTS.

4.1 Shared Code Base

PAT.GUI package implements main GUI and the plug-in architecture. *GUI* package loads the syntax files of different modules during the system initialization, which stores the syntax information (e.g., keywords, folding, and auto-completion) and module information. When users want to perform syntax checking, simulation or verification of an input model, the linked module (packed as one DLL) is loaded into the system dynamically using reflection technique and then the corresponding interface method is invoked.

PAT.Common package contains module interfaces, common GUI classes, expression classes and verification libraries. Abstract class *ModuleFacadeBase* in *PAT.Common* package defines the module interface. It behaves as the central gate communicating with the GUI package by adopting the FAÇADE design pattern, e.g., check syntax, show simulator or model checker window. All modules must implement this interface in order to be recognized by the plug-in framework. *Specification* is the interface class for the internal representation of system models. It also uses FAÇADE pattern to communicate with *SimulatorGUI* and *ModelCheckerGUI* to provide the information of the user input model. *SimulatorGUI* and *ModelCheckerGUI* are the graphic user interfaces for simulation and model checking. *Specification* is composed by system model and properties (named *Assertion*s in Figure 2) which are to be verified. Because system model information is module dependent, *Specification* is only associated with *Assertion*s in the common package.

The module interface implementation uses the ABSTRACT FACTORY pattern, where *ModuleFacadeBase* is the abstract factory, and *ModuleFacade* classes in each modules are concrete factories. The product is *Specification* interface, and the concrete products are the actual *Specification* implementations in modules. The clients of using the products are the GUI classes. The *Expression* and *Value* classes define the interfaces for a simple but general WHILE language with variables and some primitive values. Though these classes should be implemented in the actual module, we move them to the common library so that they can be shared by all modules. Variable values need to be cloned if one state has more than one outgoing transition.

Besides the module interfaces and shared libraries, *PAT.Common* package contains the intermediate layer and verification algorithms in the analysis layer. In Figure 2, we use LTS as an example. The IRL here is just one abstract class *LTSState*, which implements a single transition with a target state in LTS definition. The transition's label is stored in the *Label* property, and target state information (e.g., variable valuation, channel buffer, program counter and so on) shall be realized in the actual module since it is module related. *MoveOneStep* method generates all the possible transitions starting from the target state. *GetID* method returns the unique hashed string representation of the label and target state. If symbolic model checking is used, *EncodeProcess* method performs the encoding of the target state and outgoing transitions to Boolean formulae that can be used by the symbolic model checker or SAT solvers. The three abstract methods shall be realized in the actual module and will be used by the verification algorithms. It is clear that starting from an initial state, these methods are sufficient to generate the complete state space.

Assertion class defines the interface for properties. It has one initial state, from where it explores the system state space and produces a counterexample if the property is not satisfied. For LTS verification, four assertions are supported as shown in Figure 2. We apply the STRATEGY pattern when designing the verification algorithms. For example, for safety properties (e.g., *DeadlockAssertion*, *RefinementAssertion* and *ReachabilityAssertion*), three searching strategies are possible: depth-first-search (DFS), breadth-first-search (BFS) and symbolic verification algorithm. For liveness properties (i.e., *LTLAssertion*), two searching strategies are possible: nested DFS [18] or Tarjan's strongly-connected components searching algorithm. These strategies are selected

according to users' choice during the run-time.

Verification algorithms need to keep track of visited states (*StateManager*) or have a scheduler if the concurrent processes have different priorities (*SchedulingManager*). They are linked with the *Assertion* so that the verification algorithm can use them to store the states and perform process scheduling without worrying about the actual implementation. These two classes also adopt the STRATEGY pattern since a number of state hashing and scheduling strategies are possible.

To increase the reusability of the code, we create the generic version of popular algorithms using TEMPLATE pattern. For example, DFS and BFS are searching algorithms used in all safety property verification. To have a reusable implementation, we create a generic DFS (BFS) algorithm with an abstract early termination condition. Different algorithms implement the early termination condition differently. For instance, *DeadlockAssertion*'s early termination condition is that the current state is a deadlock state. *ReachabilityAssertion*'s early termination condition is that the current state satisfies the desired condition.

4.2 Encapsulated Module Design

Each module is encapsulated in a stand alone package, which implements the module interface and specification interface, and contains modeling layer's components, i.e., language parser (e.g., *CSPParser*), variable valuation and channel buffer (stored in *Valuation*) and language syntax classes.

The language syntax classes in CSP module naturally form a COMPOSITE pattern by following the language grammar. Each language construct (e.g., parallel composition, sequential compositions, choice process and so on) is implemented as a single class, which implements the *Process* interface with two key methods: *MoveOneStep* and *EncodeProcess*. These two methods implement the semantics of the language syntax for the two different model checking approaches. *MoveOneStep* method implements the operational semantics rules to realize the state transitions from current state to a list of target states. *EncodeProcess* performs the encoding of the syntax to Boolean formulae that can be used by the symbolic model checker or SAT solvers.

The most important class to be implemented is *CSPState* class, which realizes the concrete transitions of the LTS. For CSP#, each system state contains the current valuation of variables, buffered elements in the channels and current active process [30]. This composition is exactly implemented in Figure 2. The *MoveOneStep* method (or *EncodeState* method) in *CSPState* class will invoke the *MoveOneStep* (or *EncodeProcess*) method of the current active process by providing current variable valuation and channel buffer data. This guarantees that LTS is generated by following the operational semantics. At this point, the modeling layer is completed.

Most of the abstraction and reduction techniques are highly language dependent [20]. For example, different timed operators in real-time systems shall update the zones in different ways. Partial order reduction strategies require information about processes, global variables, dependencies of transitions, etc. This restriction makes the abstraction layer hardly fully independent from modeling layer. Most of the abstraction and reduction techniques are embedded inside method *MoveOneStep* or *EncodeState* to produce a reduced state space. In PAT3, we demonstrate these techniques by code samples and libraries so that our experiences can be reused.

5. PAT3 EXTENSIONS

In this section, we discuss the possible extensions of PAT3 and related technical challenges. These extensions allow domain experts to create customized model checkers in all levels, based on their knowledge of model checking.

5.1 Easy Translation with Module Interface

The easiest way of creating a model checker is to create a syntax rewriter from a domain specific language to an existing tool. Comparing with other model checkers, programming a language translator is straightforward in PAT3. Because PAT3 has well-defined APIs for *Specification* façade class and module language constructs, users only need to create the *Specification* model and generate the language constructs objects using these APIs. For example, *Specification* class in CSP# module offers the interfaces to create global variables, channels, processes and assertions. This approach requires no interaction with PAT3 codes. Most importantly, the target language model is automatically generated from the *Specification* class, which can guarantee syntax correctness. The usefulness of this extension has been demonstrated by the build-in translator from Promela to CSP# and the translator from UML state diagram to CSP#. Bogor's extensions for two languages (Java and Cadena) are based on the translation approach.

However, these are drawbacks for this approach. For example, the translation may not be optimal if special domain specific language features are present. In addition, reflecting analysis results (e.g., showing the counterexample trace) back to the domain model is often non-trivial.

5.2 Extensible Data Type and Library

Modules in PAT3 only support primitive types like integer, Boolean variables and arrays of integers. However, sometimes user defined data type is necessary and can simplify the model substantially. In *PAT.Common* package, PAT3 defines the interface for variable valuations, which includes state hashing methods, a *to-string* method for simulator display and a *deep-clone* method for duplicating the values. PAT3 provides the functionality to create arbitrary data type by simply creating a C# class inheriting the *Value* interface, which can be imported into the model (based on the reflection mechanism in .NET) and used as normal variables. Note that polymorphic in types is also possible in PAT3, i.e., user defined types can be nested. This extension is not novel, Boger's "language extension" uses the similar idea. To be more accurate, this extension is to support new data type rather than the extensions of modeling languages.

This extension brings two benefits. Firstly, data abstraction can be achieved by controlling the data hashing function which may reduce the state space significantly. For example, a check board can be presented by the position of pieces only. Therefore, a smaller state representation is possible by defining board using a user defined data type. Secondly, this method seamlessly links the implementation (in C#[5]) with the specification modeling (in CSP#). This link gives

[5]We use C# for simplicity of the implementation. Other languages like Java can also be linked with PAT3.

us the power to verify/test the real implementation by running the specification. For example, given a concurrent "Set" implementation, we can model the usages of the Set objects in CSP# model. This model can contains several processes accessing the Set object concurrently, which is hard to write using test cases. During the verification of CSP# model, the actually Set implementation is executed. Furthermore, PAT3 supports Microsoft Contracts specification (i.e., method pre-condition/post-condition and class invariants) in the user defined data types. During the verification of CSP# model, the contracts methods will be also checked, which gives one more level of verification.

5.3 Language Extension

In general, the input language of a model checker must be carefully designed, with preciseness, intuitiveness and efficiency in mind. A minor syntax extension or modification may require re-examination of the whole system. PAT3 facilitates such extensions by adapting parsers for the new syntax and implementing the corresponding syntax classes (refer to *PAT.CSP* package in Figure 2). For example, to support try-catch statement in CSP# module requires a new class implementing exception throw statement and a new class implementing try-catch semantics. PAT3's layered design requires minimum change in the system and all model checking algorithms developed can be reused. Knowledge about existing modules is required and a new parser may be created for the extended language features.

5.4 Property Extension

It is possible that a domain may have specialized properties and require dedicated model checking algorithms for these properties. PAT3's design allows seamless integration of new model checking algorithm and optimization techniques by simply creating a new assertion class, which inherits the base *Assertion* class and implements its APIs. In addition, the counterexample generation method needs to be customized to instruct how to produce the counterexample.

PAT3 offers a number of algorithm templates like generic DFS and BFS algorithms to help the fast algorithms development. Furthermore, supporting functions, like LTL to Büchi/Rabin/Streett automata conversion which is essential for LTL model checking, or DBM library which is for real-time system verification, are provided in PAT3. With this design, we have successfully extended PAT3 with the algorithms for divergence checking, refinement checking in real-time system module and new deadlock and probabilistic reachability checking in probabilistic systems.

5.5 Abstraction Extension

Abstraction techniques are often language dependent and hard to extend. As a result, abstraction techniques are to be encoded as part of language semantics, in the *MoveOneStep* or *EncodeProcess* methods of each language construct in each module. PAT3 offers a framework for abstract-refinement techniques. If over-approximation abstraction is applied, users must override the method to check whether a generated counterexample is spurious and override the method to refine the abstraction.

Currently, PAT3 offers one module independent abstraction, i.e., process counter abstraction. Parameterized systems are characterized by the presence of a large (or even unbounded) number of behaviorally similar processes, and they often appear in distributed/concurrent systems. A common state space abstraction for checking parameterized systems groups behaviorally similar processes rather than keeping track of all process identifiers. When a system has identical concurrent processes, process counter abstraction can be implemented by invoking the library provided in PAT3 to update the counter and mark the group of identical processes as abstracted. The verification algorithms will automatically recognize this mark and check whether the generated counterexample is spurious due to the abstraction or a real counterexample. With the provided sample codes, counter abstraction can be implemented quickly and correctly.

5.6 Module Extension

Module extension is the last resort if the previous approaches are not applicable. In this case, users need to implement module components and make connections with the intermediate layer. This approach is the most complicated compared with the ones discussed previously. Nevertheless, this approach gives the most flexibility and efficiency as users control the language syntax and assertions to be supported, and abstraction/reduction techniques to be applied. To further simplify the process of building a module in PAT3, we have developed a module generator tool[6]. By providing module name, language syntax construct names and choices of assertions, module generator can automatically generate the module project (in C#) with interface classes (e.g. *ModuleFacade*, *Specification*, state interface etc.), language classes and code skeleton. With the generated code, users only need to create a parser and implement the operational semantics of the modeling language. Module extension in PAT3 requires only knowledge about the module interfaces and IRL layer. New module can be developed independently without the source code of PAT3. This approach is still feasible for domain experts who have only the basic knowledge on model checking, since verification algorithms and reduction techniques are separated from modeling languages.

It is difficult to quantify the effort required to build a high-quality module in PAT3. We developed a Timed Automata (TA) module in PAT3 to experiment the effort. With a basic understanding of PAT3 tool design, we finished the TA module within a month. In total, users only need to implement 7 classes to create this module, i.e., 2 interface classes (generated), 1 parser class, 3 language related components (partially generated) and 1 state interface (generated). Since 1995, Uppaal [4] has been dedicated to support the modeling, simulation and verification of Timed Automata. The TA module in PAT3 offers similar functionalities with much less efforts. Note that the performance of Uppaal is better than TA module because of its sophisticated optimization techniques. Nevertheless, this case study demonstrates the benefit of layered design and adoption of design patterns. The complete work-through and code sample can be found in PAT3's user manual.

6. PERFORMANCE EVALUATION

PAT3 is designed for reusability and extensibility. However, performance is also a critical measurement for model checkers. As a result, choices between the performance and extensibility have to be made. For instance, the code for Tarjan's strongly-connected components (SCC) algorithm is

[6]The tool can be found under "Tools" in the PAT3 toolbar.

Model	#Proc	Property	PAT3(s)	SPIN(s)
Deadlockfree dining philosophy	10	deadlock-freeness	15.7	14.1
same above	12	same above	232	-
Leader election for complete network	6	LTL with weak fairness	26.7	229
same above	8	same above	726	5720
Token circulation protocol for rings	7	LTL with global strong fairness	13.7	N/A
same above	9	same above	640	N/A
5-valued *register*	2	linearizability	44.9	NA
6-valued *register*	2	same above	297	NA
Scalable Non-Zero Indicator of size 2	2	same above	322	NA
Scalable Non-Zero Indicator of size 3	3	same above	6214	NA

(A) LTS-based Model Checking

Model	#Proc	Property	PAT3(s)	Uppaal(s)	Uppaal-o(s)
Fischer's mutual exclusion protocol	4	mutual exclusion	0.05	0.09	1.18
same above	5	same above	0.15	0.19	696.4
same above	6	same above	0.68	0.81	-
CSMA/CD protocol	5	liveness LTL	0.22	0.26	39.15
same above	7	same above	2.63	6.40	-
same above	10	same above	34.3	181	-
Railway control system	6	liveness LTL	5.24	0.43	-
same above	8	same above	349	20.7	-

(B) TTS-based Model Checking

Model	#Proc	Property	PAT3(s)	PRISM(s)
Probabilistic N-process mutual exclusion	6	mutual exclusion	1.161	0.364
same above	8	same above	8.624	0.937
Probabilistic dining philosophy	5	deadlock-freeness	2.413	0.156
same above	6	same above	25.775	0.672
Shared coin randomized consensus algorithm	4	liveness LTL	0.379	21.9
same above	6	same above	5.854	1755
Probabilistic CSMA/CD protocol	2	liveness LTL	0.933	2.314
same above	3	same above	6.284	7.233

(C) MDP-based Model Checking

Table 1: Performance evaluation on PAT3's model checking algorithms

duplicated in many modules for performance reasons, which could be easily generalized as a TEMPLATE algorithm.

In the following, we evaluate the library of model checking algorithms implemented in PAT3. We compare PAT3's performance with state-of-the-art model checkers. Table 1 shows the performance of model checking algorithms based on LTS, TTS and MDP respectively. The experiment data are obtained using a PC with 2 Intel Xeon CPUs at 2.13GHz and 32GB RAM. N/A means not applicable due to either a language feature required by the system is missing or the kind of property is not supported; $-$ means either out of 32GB memory or more than 4 hours. Note that all properties are true so that all states are explored. PAT3 is available at `http://www.patroot.com`. Experiments details can be found at `http://www.comp.nus.edu.sg/~pat/system`.

Table 1(A) shows experimental results on applying LTS-based model checking algorithms to population protocols and concurrent data objects. PAT3's performance is compared with SPIN. In the first experiment with the classical dining philosopher example, SPIN marginally outperforms PAT3 for 10 philosophers. It runs out of memory for 12, whereas PAT3 finishes it in 232 seconds. The experiments on the leader election protocol and token circulation protocol show that PAT3 outperforms SPIN if fairness assumption is required. The reason is that PAT3 implements an efficient SCC-based LTL verification algorithm that can check various fairness assumptions directly. However, SPIN's nested-DFS method can only handle weak fairness. In the register and scalable non-zero indicator experiments, PAT3 performs trace refinement checking to verify linearizability of concurrent data objects, which is not supported in SPIN.

Table 1(B) shows experimental results on applying TTS-based model checking algorithms (using zone abstraction) to benchmark real-time systems. PAT3's performance is compared with the UPPAAL model checker. Three benchmark real-time systems are used. The column **Uppaal** shows the time of verifying the models using UPPAAL. PAT is faster than UPPAAL for the first two examples due to the effective zone abstraction technique. In the last example, UPPAAL outperforms PAT3 by using the effective optimization techniques named *extrapolation*. The column **Uppaal-o** shows the result of verifying the same models with *extrapolation* disabled. PAT3 outperforms **Uppaal-o** significantly. This has lead to our ongoing work on realizing *extrapolation* in PAT3's DBM package.

Table 1(C) shows experimental results on applying MDP-based probabilistic model checking algorithms to probabilistic systems. PAT3's performance is compared with the PRIS-

M model checker. We use the iterative method in calculating the probability and set termination threshold as relative difference 10^{-6} (exactly same as PRISM). In the experiments of probabilistic mutual exclusion algorithm and probabilistic dining philosopher example, PRISM outperforms PAT3. The main reason is that PRISM is based on BDD, which handle large number of states in these examples. The next two experiments, however, show that PAT3 sometimes outperforms PRISM for certain class of properties. The main reason is that PAT3 models have much less states than PRISM models thanks to the hierarchical modeling language in PAT3.

We remark that the comparison above may not be completely fair due to many reasons, e.g., difference in modeling languages or property specification languages or the model checking approaches. The above results thus should be used as an indication that PAT3 offers comparable performance to existing model checkers. At same time, we are experimenting new optimization techniques or state reduction techniques constantly, thanks to PAT3's extensible architecture. In summary, PAT3 offers a library of well-optimized model checking algorithms as well as a framework for developing new model checkers.

7. RELATED WORK

There has been a large body of work on tools supporting verification using model checking technique. We partially list the related model checkers according to the semantic models. For concurrent systems, LTS-based model checking in PAT3 is related to popular tools like SPIN [18], NuSMV [10], FDR [27], etc. In terms of real-time verification, TTS-based model checking in PAT3 is related to a number of automatic verification supports for Timed Automata, including Uppaal, KRONOS [6] and so on. For probabilistic systems, MDP-based model checking in PAT3 is related to probabilistic model checkers including PRISM and MRMC [19]. Different from these dedicated model checkers, PAT3 is more than one model checker for one language, instead, it is an extensible framework designed to facilitate development of new modeling/verification techniques.

As an extensible model checking framework, PAT3 is closely related to Bogor [14], the Model-Checking Kit [28], LTSA [24] and SAL [13]. The most relevant tool is Bogor, which supports a modeling language close to Java. Bogor's extensibility is reflected in the extensions to the user defined data types[7], verification algorithms and optimization techniques. PAT3 improves Bogor's design by separating system components using the layered design and the concept of state interface, which allows extensions in all components. The support of the 3 semantic models is far beyond the capability of Bogor. Furthermore, to develop a new module in Bogor can only base on the translation approach, which requires the new modeling language to be a sub-language of Bogor's input language. Fully customized module development based on the semantic model is not supported in Bogor. Model-Checking Kit [28] is an open model checker framework supporting extensions in languages and verification algorithms with the semantic model Petri nets. LTSA allows extension in modeling language by compiling input languages (e.g., Message Sequence Chart or Web Service) into LTS. SAL is a collection of separate tools for performing program analysis, theorem proving, and (both explicit and symbolic) model checking on transition systems. SAL's input language is an LTS-like language, which is intended to serve as the target for translators that extract the transition systems from other modeling languages. Compared with these three, PAT3's semantic model is designed as state interfaces, which supports on-the-fly verification. In Kit, LTSA and SAL, the complete model needs to be converted into the semantic model first before starting the verification. Therefore, the whole model may need to be constructed first even if the property to be checked is false. In addition to the difference mentioned above, PAT3 allows extensions for a wide range of modeling languages thanks to the three different sematic models. PAT3 is also unique that it integrates explicit and symbolic model checking uniformly, which is never addressed in these tools. Note that in SAL, the symbolic model checkers and explicit model checkers are separate tools. Hence, there is no software design involved in SAL. Lastly, as shown in the experiments section, PAT3 achieves competitive performance with the state-of-the-art model checkers, but still offers the great extensibility. Other model checking frameworks related to PAT3 include the IF TOOLSET [7] and ESMC framework [21].

A remotely related tool is Java Path Finder (JPF) [32], which is a ground-breaking model checker working directly on Java byte-code. JPF's flexibility has been demonstrated by the incorporation of a variety of search modes such as heuristic searches. Working directly on byte-code allows one to claim to some extent that the code being checked is the code that is actually going to be run. However, our goal with PAT3 is to provide an extensible modeling language that can support checking of artifacts at different levels of abstraction. Thus, PAT3 offers a compromise: it supports checking of programming language like C#, and yet, it is flexible enough to be adapted to obtain dedicated checkers for different modeling languages. For the performance, some initial experiments suggest that PAT3 is much faster than JPF, since JPF is running on the Java Virtual Machine.

The BDD library in PAT3 is designed such that users only need the minimum knowledge of BDD in order to use it. It is thus different from approaches like JTLV [25], which are designed to allow flexible control of BDD packages for advanced users. Furthermore, our library is intended to support systematic encoding of (at least a large subset of) existing compositional languages with ease.

Compared with early versions of PAT (PAT1 [31] and PAT2 [23])[8], the enhancement in PAT3 is dramatic. We restructure the architecture by introducing the 4-layered design with a clear state interface design. TTS and MDP are introduced in the IRL. Symbolic model checking is also supported in the framework. 11 modules have been developed in PAT3. PAT3 has come to a stable stage with solid testing and various applications. More than 100 built-in examples and hundreds of test cases are embedded in PAT3. It is now being used by a number of institutions as a research or educational tool.

[7]Bogor claims it supports extensions in modeling language. But in fact, its extension is same as the data type extension in PAT3. Real language extension like add multi-party synchronization in Bogor is impossible without the change of parser and source code.

[8]Note that [31] and [23] are tool papers, which present the functionalities of PAT only.

8. CONCLUSION

As model checking continues to grow in popularity, model checking tools need to adapt so that they can effectively support a broad range of system descriptions and property languages. One approach to overcoming the significant cost of model checking is to exploit available domain knowledge of specific software artifacts to develop highly-optimized state space representations, reductions and search algorithms. PAT3's extensible, customizable tool architecture will help minimize the amount of model checking-specific knowledge that a domain expert needs to build cost-effective analysis capabilities. In addition, this approach can be repeated to build similar verification framework.

To show the practicality, we have implemented 11 domain-specific model checkers using PAT3, which can give significant space and time reductions while reducing the cost of the implementation relative to other model checkers. Furthermore, experiment results show that PAT3 does verification efficiently. Our future works include integration of other model checking techniques (e.g., counterexample guided abstraction refinement, assume-guarantee model checking and verification using SMT solvers), automatic module generation based on syntax grammar and operational semantics rules as well as software component reuse to reduce the redundant code crossing different modules.

9. REFERENCES

[1] PAT: Process Analysis Toolkit. http://www.patroot.com/.

[2] SAT Competition. http://www.satcompetition.org/.

[3] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, 1987.

[4] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, Y. Wang, and M. Hendriks. UPPAAL 4.0. In *QEST 2006*, pages 125–126, 2006.

[5] A. Biere, E. M. Clarke, R. Raimi, and Y. S. Zhu. Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs. In *CAV 1999*, pages 60–71. Springer, 1999.

[6] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV 1998*, pages 546–550, 1998.

[7] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. Tools and Applications II: The IF Toolset. In *SFM*, pages 237–246, 2004.

[8] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[9] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, 1992.

[10] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In *CAV*, pages 359–364, 2002.

[11] E. M. Clarke, A. Biere, R. Raimi, and Y. S. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[12] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[13] L. M. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari. Sal 2. In *CAV 2004*, pages 496–500, 2004.

[14] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building Your Own Software Model Checker Using the Bogor Extensible Model Checking Framework. In *CAV*, pages 148–152, 2005.

[15] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[16] A. Hinton, M. Z. Kwiatkowska, G. Norman, and D. Parker. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *TACAS*, pages 441–444, 2006.

[17] C. A. R. Hoare. *Communicating Sequential Processes*. International Series on Computer Science. Prentice-Hall, 1985.

[18] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Wiley, 2003.

[19] J. Katoen, I. S. Zapreev, E. M. Hahn, H. Hermanns, and D. N. Jansen. The Ins and Outs of the Probabilistic Model Checker MRMC. In *QEST 2009*, pages 167–176, 2009.

[20] M. Kattenbelt. Towards an explicit-state model checking framework, August 2006.

[21] M. Kattenbelt, T. Ruys, and A. Rensink. An object-oriented framework for explicit-state model checking. In *VVSS 2007*, pages 84–92, 2007.

[22] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[23] Y. Liu, J. Sun, and J. S. Dong. Developing Model Checkers Using PAT. In *ATVA 2010*, pages 371–377, 2010.

[24] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. Wiley, 1999.

[25] A. Pnueli, Y. Sa'ar, and L. D. Zuck. Jtlv: A framework for developing verification algorithms. In *CAV 2010*, pages 171–174, 2010.

[26] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0, 1, infty)-Counter Abstraction. In *CAV 2002*, pages 107–122, 2002.

[27] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[28] C. Schröter, S. Schwoon, and J. Esparza. The model-checking kit. In *ICATPN 2003*, pages 463–472, Berlin, Heidelberg, 2003. Springer-Verlag.

[29] A. P. Sistla. Safety, Liveness and Fairness in Temporal Logic. *Formal Asp. Comput.*, 6(5):495–512, 1994.

[30] J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE 2009*, pages 127–135, 2009.

[31] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV 2009*, pages 702–708, Grenoble, France, June 2009.

[32] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Automated Software Engineering Journal*, pages 3–12, 2000.