Model Checking Extended Computation Tree Logic

Daniel Horgan

May 30, 2011

Abstract

Computation Tree Logic is widely used for modelling the behaviour of simple systems over time, but its expressive power is limited. More powerful logics such as CTL* and the modal mu-calculus have the disadvantage that their model-checking problems are comparatively intractable, and they can be unintuitive. A recent contribution by Axelsson et al. introduces the 'Extended CTL' family of logics, in which the Until and Release operators of CTL are parameterised by various classes of automaton. This has the advantage of increasing the expressive power of the logic, whilst (in the case of pushdown automata) preserving the tractability of model checking. Using algorithms based on those described in the paper, this project implements (to the best of my knowledge) the first concrete system for model checking CTL[PDA, DPDA], the logic in which both operators are refined by pushdown automata, which are deterministic in the case of Release. As well as a robust and tested core checking procedure, we provide a set of commands for loading and displaying automata, systems and formulas. It is possible to check both regular and pushdown systems, and checking a fixed formula is possible in time quadratic in the size of the system and sizes of the automata used.

Contents

1 Introduction

Automatic formal verification of programs is an important topic in computer science, because it is difficult to evaluate correctness properties of complex programs manually. One of the main approaches is based on *model checking*: relevant aspects of the behaviour of a program are represented as an abstract logical model, and the program specification is formulated as a set of formulas of a corresponding logic. By checking whether the model satisfies the formulas, we can discover whether the program meets the specification they embody.

Many different logics have been developed and used in this way; amongst the most popular have been the temporal logics CTL and LTL. Their formulas are evaluated over *labelled transition systems*: such systems consist of a finite set of states at which various propositions may hold, together with rules specifying the possible movements between them.

While both of these logics are useful and well-studied, they are unable to express many of the more complicated properties that we may wish to check. Moreover, labelled transition systems cannot directly be used to represent programs with infinitely many states. Since we wish to permit software with unbounded recursion, this is a severe limitation. For example, consider the case of a shared resource whose lifetime is controlled by reference-counting. We wish to check that the count is decremented exactly the same number of times as it is incremented, and that the count does not reach zero before it is safe for the resource to be released. Since the count may legitimately become arbitrarily large, a finite state system cannot be used to track this property.

More powerful logics have their own problems.[1] The model checking problem for CTL* is PSPACE-complete [1] [10]. Checking modal mu-calculus formulas on pushdown systems is EXPTIME-complete [11] – while strong steps have been made towards solving this efficiently [7], the logic is also frequently unintuitive and is unable to express non-regular properties.

To address these issues, Axelsson et al. have developed a group of new logics in which CTL is extended by classes of automata (or, equivalently, of formal languages). [1] Automata associated with a formula can refine the path quantifiers of CTL such that it is possible to specify such properties as 'there exists a path where p holds at every second step'. This is an example of a regular property, since the language of paths of even length can be accepted by a deterministic finite automaton (DFA). Properties from elsewhere in the Chomsky Hierarchy can be used depending on the classes of automaton permitted in a given CTL-extension – for example, equipping the path quantifiers with pushdown automata (PDA) allows the specification of context-free properties. The logic in which existential **Until** formulas may be refined by automata from a class $\mathfrak{A}$ and **Release** from $\mathfrak{B}$ is written CTL[$\mathfrak{A}, \mathfrak{B}$]. For a more detailed explanation of the semantics of Extended CTL, see the Background section (or, indeed, the original paper [1]).

Most interestingly from a verification standpoint, the investigation yielded some promising complexity results for certain Extended CTL model checking problems. In particular, it was shown that CTL[PDA, DPDA][2] was in P.

[1] A detailed justification for Extended CTL is given by Axelsson et al. (2010), upon which this summary is based.

[2] A DPDA is a deterministic pushdown automaton.

As well as being computationally efficient, this technique is relatively intuitive. Many programmers have a good understanding of regular expressions and context-free grammars, but the fixed-point operators which lend the modal mu-calculus its power are likely to be unfamiliar.

At least in theory, then, CTL[PDA, DPDA] has good potential as a practical logic for software verification. For this reason, and to determine to what extent this is borne out in practice, the primary aim of this project was to create a concrete, working implementation of the model checking algorithms described in Axelsson et al.

To this end, an input language was defined for specifying pushdown and non-pushdown automata and systems, as well as extended CTL formulas. A set of commands is provided for interacting with these objects, and in particular for checking formulas against systems. This is possible both when the pushdown component is in the automaton and when it is in the transition system.

The report is structured as follows. We formally define the syntax and semantics of CTL$[\mathfrak{A}, \mathfrak{B}]$, based on the description in the paper. We also recall the algorithms for model checking the Until and Release clauses set out therein, and elaborate on them with some thought to implementation. We see briefly how the logic can be used for verifying programs, by representing control-flow with pushdown systems.

Next we analyse the requirements for the concrete implementation, followed by explaining and justifying the design of the system in detail. We cover the strategies used for testing, and an example. Finally, we use the system to check a Java program, and conclude with some thoughts to future work.

We also include a concise user manual, and listings of some of the project code.

2 Background

In all of the exposition that follows, let **Prop** be a countably infinite set of proposition variables, let Σ be a finite set of action names, and let Γ be a finite set of stack symbols.

We now recall some standard definitions.

Definition 1. A deterministic finite automaton (DFA) $\mathcal{A}$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a set of states, Σ is as established above, $\delta : (Q \times \Sigma) \to \Sigma$ is a (total) transition function, $q_0 \in Q$ is the initial state, and F is the set of accepting states. We write $s \xrightarrow{a} s'$ if $((s, a), s') \in \delta$. The accepting language of the DFA is defined as $L(\mathcal{A}) = \{a_1 a_2 ... a_n \in \Sigma^* : \exists$ a path $q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} q_n$ for some $q_1 \ldots q_n \in Q$ with $q_n \in F\}$.

Definition 2. A labelled transition system (LTS) is a triple $\mathcal{T} = (\mathcal{S}, \to, l)$, where $\mathcal{S}$ is a set of states, $\to \subseteq \mathcal{S} \times \Sigma \times \mathcal{S}$ is the transition relation, and $l : \mathcal{S} \to 2^{\mathbf{Prop}}$ is a labelling function. A *path* in the LTS is a sequence $\pi = s_0, a_1, s_1, a_2, s_2, \ldots$ where $s_i \in \mathcal{S}$ and $a_i \in \Sigma$ such that $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for each $i \in \mathbb{N}$, and such that π is infinite or ends in some s_n which is not the head of any transition rule.

Definition 3. A pushdown automaton (PDA) $\mathcal{A}$ is a 6-tuple $(Q, \Sigma, \Gamma, \Delta, q_0, F)$ where Q is a set of states, Σ and Γ are as established above, $\Delta \subseteq (Q \times \Gamma \times \Sigma) \times (\Sigma \times \Gamma^*)$ is a set of pushdown transition rules, $q_0 \in Q$ is the initial state, and F is the set of accepting states. We write $\langle s, \gamma \rangle \xrightarrow{a} \langle s', \gamma' \rangle$ if $((s, \gamma, a), (s', \gamma')) \in \Delta$. The accepting language of

the PDA is defined as $L(\mathcal{A}) = \{a_1 a_2 ... a_n \in \Sigma^* : \exists$ a path $\langle q_0, \epsilon\rangle \xrightarrow{a_1} \langle q_1, \gamma_1\rangle \xrightarrow{a_2} \ldots \xrightarrow{a_n} \langle q_n, \gamma_n\rangle$ for some $q_1 \ldots q_n \in Q$ with $q_n \in F$ and $\gamma_1 \ldots \gamma_n \in \Gamma^*\}$. If Δ is a total function $(Q \times \Gamma \times \Sigma) \to (\Sigma \times \Gamma^*)$ then we say the PDA is *deterministic* – it is a DPDA.

Definition 4. A pushdown system (PDS) is a 4-tuple $\mathcal{P} = (\mathcal{S}, \Gamma, \to, l)$ where $\mathcal{S}$ is a set of states, Γ is as above, $\to \subseteq (Q \times \Gamma \times \Sigma) \times (\Sigma \times \Gamma^*)$ is a set of pushdown transition rules, and $l : \mathcal{S} \times \Gamma \to 2^{\mathbf{Prop}}$ is the labelling function. The PDS is *deterministic* if $\to$: $(Q \times \Gamma \times \Sigma) \to (\Sigma \times \Gamma^*)$ is a total function. A *path* in the PDS is a sequence $\pi = \langle s_0, \gamma_0\rangle, a_1, \langle s_1, \gamma_1\rangle, a_2, \langle s_2, \gamma_2\rangle, \ldots$ such that $\langle s_i, \gamma_i\rangle \xrightarrow{a_{i+1}} \langle s_{i+1}, \gamma_{i+1}\rangle$ for each $i \in \mathbb{N}$, and such that π is infinite or ends in some s_n which is not the head of any transition rule.

2.1 Extended Computation-tree logic

Definition 5. The syntax for an CTL$[\mathfrak{A}, \mathfrak{B}]$ formula is defined recursively as follows:

$$\phi ::= q | \phi \wedge \phi | \neg\phi | \mathtt{E}(\phi \mathtt{U}^{\mathcal{A}} \phi) | \mathtt{E}(\phi \mathtt{R}^{\mathcal{B}} \phi)$$

where $q \in \mathbf{Prop}$, $\mathcal{A} \in \mathfrak{A}$, and $\mathcal{B} \in \mathfrak{B}$.

Definition 6. Formulas are evaluated with respect to states of an LTS $\mathcal{T} = (\mathcal{S}, \to, l)$, according to the following semantics:

$\mathcal{T}, s \models q$ iff $q \in l(s)$ $\mathcal{T}, s \models q$ iff $q \in l(s)$
$\mathcal{T}, s \models \phi_1 \wedge \phi_2$ iff $\mathcal{T}, s \models \phi_1$ and $\mathcal{T}, s \models \phi_2$
$\mathcal{T}, s \models \neg\phi$ iff $\mathcal{T}, s \not\models \phi$
$\mathcal{T}, s \models \mathtt{E}(\phi_1 \mathtt{U}^{\mathcal{A}} \phi_2)$ iff $\exists$ a path $\pi = s_0, a_1, s_1, \ldots$ with $s_0 = s$ and $\exists n \in dom(\pi)$ s.t. $a_1 \ldots a_n \in L(\mathcal{A})$ and $\mathcal{T}, s_n \models \phi_2$ and $\forall i < n : \mathcal{T}, s_i \models \phi_1$
$\mathcal{T}, s \models \mathtt{E}(\phi_1 \mathtt{R}^{\mathcal{A}} \phi_2)$ iff $\exists$ a path $\pi = s_0, a_1, s_1, \ldots$ with $s_0 = s$ and $\forall n \in dom(\pi) : a_1 \ldots a_n \notin L(\mathcal{A})$ or $\mathcal{T}, s_n \models \phi_2$ or $\exists i < n$ s.t. $\mathcal{T}, s_i \models \phi_1$

If $s \in \mathcal{T}$ and $\mathcal{T}, s \models \phi$ we say that s *satisfies* ϕ.

In fact, **it is also possible to evaluate formulas w.r.t. configurations of a PDS**, using directly analogous semantics – simply consider paths between configurations instead of just between control states.

2.2 Model Checking

Given a formula and a labelled transition system, the global model checking problem consists of determining which states of the system satisfy the formula.

As with standard CTL, the global model-checking problem for Extended CTL admits a polynomial-time dynamic programming solution. Specifically, the problem of checking a formula can be broken into sub-problems of checking each of the sub-formulas. In dynamic programming, one achieves an efficient solution by solving the sub-problems in topological order; in our case this simply means checking the sub-formulas in a 'bottom up' manner.

For the clauses other than **Until** and **Release**, obvious checking algorithms follow directly from the semantics. Checking path quantifiers in polynomial time is more complicated, and

it is this that makes up the bulk of the project. Axelsson et al. suggest algorithms for these clauses [1] – we shall recap and expand upon these in the Design section.

It turns out to be the case that it is possible to use an almost identical method for checking a CTL[DFA,DFA] formula against a deterministic PDS as for checking a CTL[PDA,DPDA] formula against a regular LTS. The only difference is in the details of how we construct the product system when checking the path quantifiers. Since both of these options are useful, both are implemented by the model checker.

2.3 Programs as Pushdown systems

CTL extended by pushdown automata is an effective choice for verification of real programs because the control flow of a recursive program can naturally be modelled by a pushdown system. In this case, the stack of the pushdown system corresponds directly to the call stack of the program.

3 Requirements

The aim of this project was to create a complete system for model-checking Extended CTL for the pushdown (i.e. context-free) case.

A number of key requirements were identified, and these goals helped to direct the design process.

3.1 Correctness

The first priority for the system was – perhaps obviously – that of correctness. Since one of the primary applications of model-checking in general is the analysis and verification of other programs, it is vital that an implementation produce accurate results. Mainstream adoption of formal verification methods has been slow [9], and if these techniques are ever to be more widely used, they must in the first place be reliable.

The key algorithms used should therefore be proven correct, and care must be taken to ensure that the implementation reflects the abstract version. Ultimately though, establishing confidence in the checking procedures will require extensive testing.

3.2 Usability

Using the system should be simple and practical. The system must accept problem specifications in a clear format, and produce results in an intelligible manner. If a state is found to satisfy an existence formula, a trace should be provided as evidence. This is important for the software verification use case, since the knowledge that a program does not meet a specification is not very useful without some indication as to why this is.

3.3 Efficiency

Less important than accuracy, but nevertheless desirable is that the system should be computationally efficient.

Naturally, the size of the models which can be checked will be limited by the amount of memory available. Since real-world applications of the system will involve models generated from analysis of potentially large pieces of software, we may desire to check very large systems. Therefore, the algorithms used should be space-efficient, and the implementation should not be wasteful.

Time-efficiency is of equal importance: since the key advantage of formal verification is that it saves time by finding problems which would otherwise only be caught by extensive testing, a tool's usefulness can easily be undermined if it takes a long time to produce results.

3.4 Integration

A secondary goal is to provide some means of applying the model-checker to pushdown systems modelling the control flow of real programs. Most software is complex enough that manually creating such systems would be neither practical nor reliable. Hence there is a need for a way of producing the appropriate pushdown system automatically from program source code.

4 Design

*Note: the system implementation is given the name **MCECTL** (Model Checker for Extended CTL), and we use this name to refer both to the system as a whole and the core model checking module.*

The design process essentially followed a top-down model. It was clear from the beginning that certain specific sub-systems would be required:

- **Input**: some way of reading model checking problems into the system
- **Core**: the actual model checking algorithms
- **Output**: printing results and traces obtained from checks

The data structures used in the checking process would be central to all of these systems. In particular, since the model checking algorithms are highly automata-theoretic, it was clear that an important component of the system would need to be a set of automata classes with support for certain necessary operations. In addition to basic storage of and access to states and rules, three more significant automata-related features were required:

- The loading of automata explicitly given in text form
- The construction of a minimised DFA from a regular expression
- The computation of the predecessor configurations of a set of configurations in a pushdown system

Unfortunately, investigation found no single pre-existing library of automata classes which would fulfill all of the necessary requirements. However, it was found that at least the latter two of the above features *were* already implemented – by two separate libraries. `libfa`[3] is provided as part of the **Augeas** project, and is able to create DFAs from regular expressions. `wpds`[4] is used as part of the **MOPED** model checker, and is able to efficiently compute predecessor configurations in pushdown systems.

Thus it was decided that the best approach would be to develop new classes for the various types of automata and systems, and use the various libraries by converting between the different representations as necessary. This was perhaps less elegant than an entirely self-contained solution would have been, but was a pragmatic approach – since the focus of the project was the new model checking algorithm, it made sense to use robust and well-tested external code where possible, rather than re-implement these algorithms from scratch.

With this in mind, C++ was chosen for this project because it allows for linkage with `libfa` and `wpds` (which are written in C) whilst permitting an object-oriented approach.

4.1 Systems and Automata

Using C++ templates, it was possible to create the four necessary automaton[5] types (DFA, PDA, LTS, PDS) as a single class (with two template parameters). The class is parameterised by *state type* and *action type*. Using the plain `State` class results in automata; using `KripkeState` produces systems, with labelled states. Similarly, `RegularAction` and `PushDownAction` provide the different types of transition rule used.

This approach reduced code duplication and increased flexibility; for example, to create the product system used for the Until checking, it was sufficient to change the state type to a new `ProductState` class, acting as a pair `(State, KripkeState)` drawn from the Cartesian product.

4.2 Configuration Space

While the control states of the automata are stored explicitly, the actual configurations are not[6]; instead, an automaton has an associated 'configuration space', which is used whenever it is necessary to directly make reference to configurations. The configuration space stores the names of control states and of stack symbols, and associates each pair thereof with a unique integer. This ID number is used to refer to states and configurations in the transition rules and elsewhere, so that potentially long name strings do not need to be copied unnecessarily. It also simplifies the construction of product systems.

[3] `http://augeas.net/libfa/`

[4] `http://www.fmi.uni-stuttgart.de/szs/tools/wpds/`

[5] For brevity, the term 'automata' is taken here also to include transition systems

[6] Here 'configuration' refers to a combination of a control state and a top stack symbol.

4.3 Input

The input language is tokenised and parsed using **flex** and **Bison** respectively. See the Appendix for a detailed explanation of the input language. The **Boost.Spirit** parsing framework was considered as an alternative to the flex/Bison combination. This might have resulted in a more elegant input system; however flex/Bison had the advantage of being well-known and comparatively easy to use, once set up. Their idiosyncrasies are also better documented than those of Spirit.

An Abstract Syntax Tree is constructed recursively during the parse; ultimately the input text is transformed into a series of Command objects. This helps to modularise functionality, and means that the actual execution of the commands can be delayed. As well as generally adding flexibility, this decision means that it is possible to ensure definitions are well-formed before they are undertaken.

In the case of formulas, the intermediate representation as an AST has another advantage: it affords an opportunity to rewrite clauses into another form. This allows us to provide model checking procedures for only a minimal portion of the logic; other types of formula are converted to this according to the usual equivalences.

In particular, checking procedures are implemented for the base cases true and false, propositional variables, negation, conjunction, $\mathtt{E}(\phi \mathtt{U} \psi)$ and $\mathtt{E}(\phi \mathtt{R} \psi)$.

Disjunction and implication are handled by transforming the AST as follows:

$$\phi \vee \psi \equiv \neg(\neg\phi \wedge \neg\psi)$$

$$\phi \rightarrow \psi \equiv \neg(\phi \wedge \neg\psi)$$

By a similar duality, we can define the universal path quantifiers: [1]

$$\mathtt{A}(\phi \mathtt{U} \psi) \equiv \neg\mathtt{E}(\neg\phi \mathtt{R} \neg\psi)$$

$$\mathtt{A}(\phi \mathtt{R} \psi) \equiv \neg\mathtt{E}(\neg\phi \mathtt{U} \neg\psi)$$

and for `EX`, `AX` and similar – see Axelsson et al. for details.

The conversion from AST to fully instantiated object is carried out by the DeclareAutomatonCommand, `DeclareRegexCommand` and `DeclareFormulaCommand` classes. These each use the Visitor pattern to iterate over the structure of the AST of the relevant object and incrementally construct the full object. Checks for validity are performed during this process.

4.3.1 Regular Expressions

Since it can be tedious to define automata manually, it is also possible to specify a DFA using a regular expression. The regex is parsed into its own AST – as with the other structures – but when interpreted, libfa is used to construct a DFA with the name given. The decision to use libfa was taken because to reimplement the necessary minimisation algorithms would have been time-consuming and potentially less robust. However, libfa naturally uses its own representation for automata and regular expressions, so we need to convert between them to retrieve the results. This is done by the `DeclareRegexCommand` class.

4.4 User interface

Given the usability requirements, a traditional read-eval-print loop was an obvious choice as a front-end to the system. The REPL makes use of the **GNU readline** library, so that standard keyboard shortcuts can be used for such operations as retrieving previously entered commands from the history, and auto-completing filenames.

A full graphical user interface would have provided an even greater level of usability, but was judged to be beyond the scope of this project. However, since the main core of the model checker is built as a separate (static) library, it would be easy to extend the system with an alternative front-end.

A typical session involves loading system and formula definitions from a file, followed by specifying checks to run against these objects. It is also possible to display the automata and systems that have been loaded.

The interpretation of the command text, whether input directly or loaded from a file, is handled by the `CommandParser`, which uses flex and Bison to convert input text into `Command` objects. These objects are executed by the `CommandProcessor`, which maintains a reference to the `Environment`.

4.5 Environment

The environment is used to store all of the automata, systems, and formulas that have been loaded at any given point in the program. It also stores the results of model-checks, so that it is only necessary to check a formula once – to recall the results again, they are simply looked up in the environment. This has the potential to save computation time if the user wishes to check several similar formulas against a system. It also allows the user a greater degree of interactivity; it's possible to define and check a new formula during a session, without needing to reload the transition system.

4.6 Showing objects - text

The `:show` command has two purposes – with no parameter, it outputs the names of all automata, systems, and formulas that have been loaded; if provided with the name of an object, it will output a textual representation of that object via the `ToString()` method. These objects all inherit from the `Showable` abstract class.

4.7 Showing objects - GraphViz

The `:xshow` command provides a way of displaying loaded automata and systems graphically. It achieves this by obtaining a **GraphViz** 'dot' format representation of the requested object via the `ToDot()` method, and piping the text to the `dot` program. `dot` must be in the system PATH for this to work. This is somewhat rudimentary, but the feature is extremely helpful and contributes a great deal to usability.

4.8 Model checking algorithm

The dynamic programming aspect of the procedure is handled in a fairly standard way: we store a 'table' of results for formulas in the Environment, and the entries are filled in on demand. (In fact, for fast lookup, results are stored in a `CheckResults` object, which contains a map from unique formula ID to the actual `Result` object.) This implies a recursive approach to iterating over sub-formulas.

We use the Visitor pattern once more to traverse the structure of the formula being checked. This allows us to break up the code with a separate method for checking each type of clause, and the check action is dispatched according to the class of the formula.

4.8.1 Basic clauses

Checking the non-quantifier clauses is straightforward. The general process is to first retrieve the results for any sub-formulas (these will be calculated, recursively, if necessary) – and then to iterate over the states of the system being checked, combining the corresponding results as appropriate.

For example, in pseudo-code, the procedure for checking a Conjunction clause $\phi = x \wedge y$ is as follows:

```
x_results = Check(x)
y_results = Check(y)
phi_results = []
for i in Configurations(system):
   phi_results[i] = x_results[i] && y_results[i]
SetResults(phi, phi_results)
```

In fact, all of the clauses follow this basic pattern, but they differ in how they use the sub-results. (Base cases, such as `True` and `PVar`, do not perform sub-checks, rather they simply fill in the results, as you would expect.)

4.8.2 Until

Suppose we wish to check $\mathtt{E}(x\mathtt{U}^{\mathcal{A}}y)$ against an LTS $\mathcal{T} = (\mathcal{S}, \rightarrow, l)$, where $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Axelsson et al. reduce the problem to a reachability check against a PDS constructed as a product of $\mathcal{A}$ and $\mathcal{T}$. [1]

Specifically, we construct $\mathcal{A}_{\mathcal{T}} = (Q \times \mathcal{S}, \Gamma, \rightarrow, l)$ by $\langle (p, s), \gamma \rangle \xrightarrow{a} \langle (q, t), w \rangle$ iff $\exists a \in \Sigma$ such that $s \xrightarrow{a} t$ and $\langle p, \gamma \rangle \xrightarrow{a} \langle q, w \rangle$ and $x \in l(s)$.

This PDS encodes the semantic requirement for the system and the automaton to be 'run in step' – recall that **Until** requires the existence of an alternating sequence of states and actions, and also that the first sub-formula, x, holds at each point.

In addition to this, however, a valid path's actions must be *accepted* by the automaton, and must reach a state of the system in which the *second* subformula holds. The next step,

Definition 7. Given a PDS $\mathcal{P}$, a $\mathcal{P}$-automaton is a 5-tuple $(Q, \Gamma, \delta, P, F)$ where Q is a set of states, Γ is the stack alphabet as usual, $\Delta \subseteq (Q \times \Gamma) \times Q$ is a transition function, $P \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of accepting states. We define the acceptance condition as follows:
Let $\rightarrow \subseteq Q \times \Gamma^* \times Q$ be the smallest relation such that

- $q \xrightarrow{\epsilon} q$ for all $q \in Q$
- $(\langle q, \gamma \rangle, q') \in \Delta \implies q \xrightarrow{\gamma} q'$, and
- $(q \xrightarrow{w} q' \text{ and } q' \xrightarrow{\gamma} q'') \implies q \xrightarrow{w\gamma} q'$

The automaton **accepts** configuration $\langle p, w \rangle$ of $\mathcal{P}$ if $p \xrightarrow{w} q$ for some $p \in P$ and $q \in F$.

Figure 1: $\mathcal{P}$-automaton definition, based on Esparza et al. 2000

then, is to consider the set of configurations of the product system in which these criteria are satisfied:

$$R := \{\langle (p, s), w \rangle : p \in F \text{ and } y \in l(s), w \in \Gamma^*\}$$

Finally, observe that a valid **Until** path from a state s exists iff there is a path in the product system from $\langle (q_0, s), \epsilon \rangle$ to one of the configurations in R (since automaton simulation begins with the initial state and an empty stack). This holds exactly if $\langle (q_0, s), \epsilon \rangle$ is in the *predecessor configurations* of R. The proof is straightforward – see Axelsson et al. 2010 [1] for the details. We shall refer to $\langle (q_0, s), \epsilon \rangle$ as the 'bottom configuration' for s.

The final requirement, then, is to find an efficient way of computing this set of predecessor configurations: the configurations of the product system from which, by repeatedly applying transition rules, it is possible to reach a configuration from the original set. Fortunately, this is well-studied – Axelsson et al. refer to the original method of Bouajjani et al. [3] For the implementation, though, we apply `wpds`, which uses an improved method due to Esparza et al. [6] This process is involved, but the outline is as follows: the initial set of configurations is represented symbolically by a new '$\mathcal{A}_\mathcal{T}$-automaton'[7] (See fig. 1), to which a `pre*` algorithm is applied. `pre*` is a *saturation process* – it adds new edges to the $\mathcal{A}_\mathcal{T}$-automaton until it accepts all of the predecessors.

Creating an automaton which accepts the set R is straightforward. Define a $\mathcal{A}_\mathcal{T}$-automaton $\mathcal{R} = (Q \times \mathcal{S}, \Gamma, \Delta_\mathcal{R}, P_\mathcal{R}, F_\mathcal{R})$ with $\Delta = \{(\langle (p, s), \gamma \rangle, (p, s)) : (p, s) \in Q \times \mathcal{S} \text{ with } p \in F \text{ and } y \in l(s)\}$, and $P_\mathcal{R} = F_\mathcal{R} = Q \times \mathcal{S}$. It is clear from the definition of the acceptance condition that $\mathcal{R}$ accepts R.

Applying the `pre*` procedure of `wpds` yields $\mathcal{R}_{pre^*}$: to check whether a state s satisfies the **Until** formula, we simply check whether its bottom configuration is accepted by $\mathcal{R}_{pre^*}$.

In the case that it is, we can use the `wPathFind` method of `wpds` to retrieve a witnessing trace – refer to the source code for details.

4.8.3 Release

In Axelsson et al. (2010) the problem of checking $\mathtt{E}(x \mathtt{R}^\mathcal{A} y)$ against an LTS $(\mathcal{S}, \rightarrow, l)$ is reduced to evaluating the LTL formula $\mathtt{F}p_b$ on a constructed pushdown system $\mathcal{A}_\mathcal{T} = (Q \times \mathcal{S} \cup \{g, b\}, \Gamma, \Delta, l')$, where

[7] In the terminology of Bouajjani et al. this is a $\mathcal{A}_\mathcal{T}$ multi automaton.

1. Check both subformulas
2. Explicitly construct the product system $\mathcal{A}_\mathcal{T}$
3. Construct a $\mathcal{A}_\mathcal{T}$-automaton recognising the configurations which are accepting and also satisfy the second sub-formula.
4. Apply the `pre*` algorithm to the automaton.
5. For each state of the system: check whether the new automaton accepts the state's 'bottom configuration'.

Figure 2: Until checking algorithm

$$l'(s) = \begin{cases} \{p_b\} & \text{if } s = b \\ \{\} & \text{if } s = g \\ l(s) & \text{otherwise} \end{cases}$$

and

$$((p,s),\gamma) \to \begin{cases} (g,\epsilon) & \text{if } s \in l'(x) \text{ and} \\ & (p \in F \text{ implies } s \in l'(y)) \\ (b,\epsilon) & \text{if } p \in F \text{ and } s \notin l'(y) \\ ((q,t),w) & \text{if neither of the above match} \\ & \text{and there exists } a \in \Sigma \text{ s.t.} \\ & s \xrightarrow{a} t \text{ and } (p,a,\gamma) \to (q,w) \\ & \text{for some } \gamma \in \Gamma, w \in \Gamma^* \end{cases}$$

We refer to this system (as we do in the code) as a `ReleaseSystem`, to distinguish it from the `ProductSystem` used in the **Until** checking.

To check the LTL formula, we use a method based on the one described in Esparza et al. (2000) [6].

1. Construct a Büchi automaton $\mathcal{B}$ with initial state q_0 corresponding to the negation of the formula
2. Compute the Büchi pushdown system $\mathcal{BP}$ as the product of $\mathcal{B}$ and the pushdown system
3. Compute the set of repeating heads R of $\mathcal{BP}$
4. Construct an automaton $\mathcal{A}$ accepting $R\Gamma^*$.
5. Compute $pre^*(R\Gamma^*)$

A configuration $\langle p, w\rangle$ violates the formula iff $\mathcal{A}_{pre^*}$ accepts $\langle (p, q_0), w\rangle$.

Figure 3: Checking an LTL formula against a pushdown system: method from Esparza et al. (2000) [6]

But since we only need to be able to check the specific type of LTL formula that occurs here, it is possible to make some simplifications.

The first step in checking $\mathtt{F}p_b$ is to construct a Büchi automaton corresponding to the negation of the formula. This will have one state, which is accepting, and has a self-loop for every valuation except those in which p_b is true.

$$\mathcal{B} := (\Sigma_\mathcal{B}, Q_\mathcal{B}, \delta_\mathcal{B}, q_{0_\mathcal{B}}, F_\mathcal{B})$$

where

$$\begin{array}{cc} \Sigma_\mathcal{B} := 2^{\mathbf{Prop}} & Q_\mathcal{B} := \{*\} \\ \delta_\mathcal{B} := \{* \xrightarrow{v} * : v \in 2^{\mathbf{Prop}\backslash\{p_b\}}\} & q_{0_\mathcal{B}} := * \\ & F_\mathcal{B} := \{*\} \end{array}$$

('*' is used as the name of the single state, since this is arbitrary)

Following the standard definition [6], the product of this Büchi automaton with our pushdown system, then, is a Büchi pushdown system given by:

$$\mathcal{BA}_\mathcal{T} = ((Q \times \mathcal{S} \cup \{g, b\}) \times Q_\mathcal{B}, \Gamma, \Delta', G)$$

where

$$G = \{(p, *) : p \in Q \times \mathcal{S} \cup \{g, b\}\}$$

and

$$\langle (p, *), \gamma \rangle \to \langle (p', *), w \rangle \in \Delta' \text{ iff } \langle p, \gamma \rangle \to \langle p', w \rangle, * \xrightarrow{\sigma} * \text{ and } \sigma \subseteq l'(\langle p, \gamma \rangle)$$

Noting that $* \xrightarrow{\sigma} *$ iff $\sigma \subseteq \mathbf{Prop}\backslash\{p_b\}$ we obtain

$$\langle (p, *), \gamma \rangle \to \langle (p', *), w \rangle \in \Delta' \text{ iff } \langle p, \gamma \rangle \to \langle p', w \rangle, \text{ and } \mathbf{Prop}\backslash\{p_b\} \cap l'(\langle p, \gamma \rangle) \neq \emptyset$$

or equivalently

$$\langle (p, *), \gamma \rangle \to \langle (p', *), w \rangle \in \Delta' \text{ iff } \langle p, \gamma \rangle \to \langle p', w \rangle, \text{ and } \{p_b\} \neq l'(\langle p, \gamma \rangle)$$

But note further that $\{p_b\} \neq l'(\langle p, \gamma \rangle)$ iff $p \neq b$ by construction of the pushdown system. In fact, we see that there are no rules with b as a head state – so finally we simply have:

$$\langle (p, *), \gamma \rangle \to \langle (p', *), w \rangle \in \Delta' \text{ iff } \langle p, \gamma \rangle \to \langle p', w \rangle$$

and clearly in fact $\mathcal{BA}_\mathcal{T} \cong \mathcal{A}_\mathcal{T}$ via $(p, *) \mapsto p$.

The next step is to compute the repeating heads of $\mathcal{BA}_\mathcal{T}$.

This is performed in `WPDSRelease::ComputeRepeatingHeads()`. Following Esparza et al. 2000 once more, we achieve this by computing the strongly-connected components of a head reachability graph. However, the construction of this graph is simplified by the observation that all of the states of the Büchi product system are accepting.

We also use the fact that in our implementation, all transition rules must be either pop, rewrite, or push actions. We use the **Boost.Graph** library to store the graph.

Once we have the reachability graph, we use Tarjan's algorithm to find the strongly-connected components – Boost.Graph has an implementation of this. For a head to be repeating, it needs to be in a strongly-connected component which contains an edge. This holds if the component has multiple vertices, or if it has a single vertex with a self-loop. We also add vertices which have no successors in the reachability graph, excluding b. This gives us a set of head configurations R from which it is possible to avoid ever encountering

1. $E = \emptyset$
2. $\mathcal{R} = pre^*(\{\langle s, \epsilon\rangle : s \in (Q \times \mathcal{S} \cup \{g, b\})\})$
3. For each rule in Δ:
4. Case $\langle p, \gamma\rangle \to \langle p', \epsilon\rangle$: // Pop rule
5. $E = E \cup \{(\langle p, \gamma\rangle, \langle p', \epsilon\rangle)\}$
6. Case $\langle p, \gamma\rangle \to \langle p', \gamma'\rangle$: // Rewrite rule
7. $E = E \cup \{(\langle p, \gamma\rangle, \langle p', \gamma'\rangle)\}$
8. Case $\langle p, \gamma\rangle \to \langle p', \gamma''\gamma'\rangle$: // Push rule
9. $E = E \cup \{(\langle p, \gamma\rangle, \langle p', \gamma''\rangle)\}$
10. For all p'' such that $(p', \gamma'', p'') \in \mathcal{R}$:
11. $E = E \cup \{(\langle p, \gamma\rangle, \langle p'', \gamma'\rangle\}$
12. return E

Figure 4: Constructing the reachability graph efficiently

the control state b. If it is possible to reach one of these configurations, clearly $\mathtt{F}p_b$ does not hold.

It remains only to compute the predecessors of the configurations whose heads are in R. We can do this by applying pre^* to an automaton which accepts $R.\Gamma^*$. This is implemented in `WPDSRelease::ConstructFA()`.

Hence our final algorithm is (in pseudo-code):

1. Check the sub-formulas
2. Construct the Release pushdown system, as above
3. Construct an automaton recognising the 'stack bottom' configurations
4. Compute their predecessor configurations
5. Create a reachability graph
6. Find its strongly connected components
7. A vertex is a repeating head if it is in a component with an edge, or it has no successors.
8. Construct an automaton recognising the repeating heads
9. Compute their predecessor configurations
10. For each state, check whether its bottom configuration is a predecessor

Figure 5: Release checking algorithm

4.9 WPDS Wrapper

Since libwpds is written in C and so has a non-object oriented interface, a wrapper class was used to simplify the code and prevent unnecessary access to the internal data structures. The library must also be initialised and de-initialised before and after use; using a wrapper means this can be performed conveniently via the constructor/destructor methods. In the case of release checking, it is necessary to compute two sets of predecessor configurations, so two instances of the wrapper class are used simultaneously – for this reason, a count of the active instances of libwpds is kept, so that initialisation and de-initialisation need only be performed once.

4.9.1 Complexity

Consider an Until clause, $\mathtt{E}(\phi_1 \mathtt{U}^{\mathcal{A}} \phi_2)$. Let n_A be the size of the state set of $\mathcal{A}$, and r_A the number of rules. Similarly let n_S be the number of states of the system being checked, and r_S the number of rules.

Constructing the product system takes time proportional to $n_A n_S|\Gamma| + r_A r_S$. Creating the automaton to recognise R takes time $n_A n_S|\Gamma|$. Computing pre^* takes $r_A r_S n_A^2 n_S^2 |\Gamma|^2$. (See Esparza et al. 2000 [6]) Checking which configurations are recognised by the new automaton takes $n_S^2 n_A^2 |\Gamma|$ in the worst case. Hence the complexity is dominated by calculating the predecessor configurations; Until checking is $O(r_A r_S n_A^2 n_S^2 |\Gamma|^2)$

Now consider Release; $\mathtt{E}(\phi_1 \mathtt{R}^{\mathcal{A}} \phi_2)$, and let us use the same notation. Constructing the product system is again $n_A n_S|\Gamma| + r_A r_S$ time. Constructing the automaton that recognises the bottom configuration for each control state takes time $n_A n_S$. Computing their predecessors takes time $n_A^2 n_S^2 r_A r_S$. Creating the reachability graph takes $r_A^2 r_S^2 |\Gamma|^2$ in the worst case (i.e. when all rules are push rules). Finding the SCCs is $O(|V| + |E|)$ so is bounded by $n_A n_S|\Gamma| + n_A^2 n_S^2 |\Gamma|^2$. Retrieving the repeating heads is time proportional to $n_A n_S|\Gamma|$ (the maximum number of components), and constructing an automaton to recognise them is then $n_A n_S|\Gamma|^2$. Computing its predecessors takes $n_A^2 n_S^2 r_A r_S$ since the size of the automaton won't be larger than the number of control states of the product system, which has at most of the order $r_A r_S$ rules. Finally, checking which configurations are predecessors takes no longer than $n_S^2 n_A^2 |\Gamma|$.

We can see that the overall worst-case complexity for Release is then $O(r_A^2 r_S^2 n_A^2 n_S^2 |\Gamma|^2)$.

For any formula ϕ then, which refers to automata $A_1, A_2, \ldots, A_n$, checking ϕ against a system with states S and rules Δ has worst-case time complexity:

$$O(|\phi||\Delta|^2 |S|^2 |\Gamma|^2 \prod_{i=1}^{n} |A_{i_Q}|^2 |A_{i_\Delta}|^2)$$

5 Testing

Since correctness of the checking procedure is of utmost importance, a systematic test strategy was developed. This included unit testing of some subsystems, as well as extensive black-box testing of the whole system.

For the unit tests, the **Boost Unit Test Framework** was used, since this granted a direct interface to the classes being tested. The black-box tests, in contrast, concerned only the input and ultimate output. For this reason, a separate script was written in Perl, which sends problems to the model checker via standard input, and then verifies that the received output matches the expected output. This script uses the standard Perl **Test::More** test harness.

5.1 Unit tests

Unit tests were used primarily for the part of the system which dealt with parsing. The tests each checked that the AST resulting from parsing some string matched the expected one

(or failed to parse, if the string was invalid). These tests were helpful during development for preventing regressions as new features were added.

Example:

```
test_case_t("FORMULA foo  E(hungry U[dfa] eat) ",
         "FORMULA foo  [[PVAR hungry] UNTIL AUTOMATA dfa [PVAR eat]] "),
```

checks that an **Until** formula parses.

Example:

```
test_case_t("REGULAR foo  (toast|coffee)* ",
         "REGULAR foo  [KLEENE [[ACTION toast] UNION [ACTION coffee]]] "),
```

checks a regular expression's AST.

See the appendix for a full list.

5.2 Black Box tests

The black-box tests are specified in self-contained Perl hash objects, and are all evaluated independently, using different sessions of MCECTL. The results for each state are extracted using a regular expression, and compared with those expected.

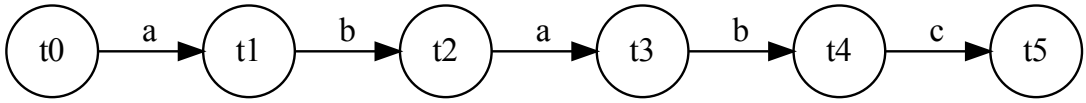
Example:

```
{
   name     => "2. LTS, five states, pushdown with one symbol",
   formula  => { name => "phi8", formula => "E( 1 U[a8] p )" },
   system   => {
      type => "LTS",
      name => "t13",
      states => ['t0 : ' ,'t1 : ' ,'t2 : ' ,'t3 : ' ,'t4 : ' ,'t5 : p'],
      rules => [
         'a:t0->t1' ,'b:t1->t2' ,
         'a:t2->t3' ,'b:t3->t4' ,'c:t4->t5'
      ]
   },
   automata => [
      {
         type => "PDA",
         name => "a8",
         states => [ 's1', '*s2' ],
         rules => [
            'a: s1[_] -> s1[PUSH s]',
            'b: s1[s] -> s1[POP]',
            'c: s1[_] -> s2[REWRITE _]'
         ]
      }
   ],
```

```
   expected => { t0 => 1, t1 => 0, t2 => 1, t3 => 0, t4 => 1, t5 => 0}
}
```

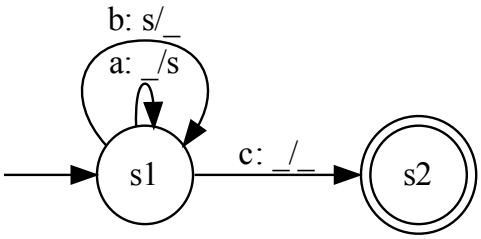
This is one of the cases for testing **Until** checking.

The transition system:



where p holds at `t5` only.

The pushdown automaton:



The formula: $\phi =$ `E`(true `R`$^{\mathcal{A}}$ p), where $\mathcal{A}$ is the above automaton.

It is clear that the automaton accepts the language { (ab)*c }. States t1, t3 and t5 have no initial a-transition, so there can be no accepting path to a p-state from these. States t0, t2 and t4 do have such paths though, as $(\text{t0}, \text{s1}) \xrightarrow{a} (\text{t1}, \text{s1}) \xrightarrow{b} (\text{t2}, \text{s1}) \xrightarrow{a} (\text{t3}, \text{s1}) \xrightarrow{b} (\text{t4}, \text{s1}) \xrightarrow{c} (\text{t5}, \text{s2})$ in the product system.

Hence t0, s2, s4 are expected to be satisfying, whereas the other states are not. This is specified in the test case.

The results returned by the model checker are:

```
Results:
   t0: T     [ <> s(s1,t0) --a--> <_> s(s1,t1) --b--> <> s(s1,t2) --a-->
               <_> s(s1,t3) --b--> <> s(s1,t4) --c--> <> s(s2,t5)__ ]
   t1: F
   t2: T     [ <> s(s1,t2) --a--> <_> s(s1,t3) --b--> <> s(s1,t4) --c--> <> s(s2,t5)__ ]
   t3: F
   t4: T     [ <> s(s1,t4) --c--> <> s(s2,t5)__ ]
   t5: F
```

so the test passes.

See the appendix for a full list of tests.

6 Application to Checking Java Programs

A similar project which performs this task is Matthew Hague's **PDSolver**[8] [7]. The focus of PDSolver is checking modal mu-calculus properties; however, it includes a **JimpleToPDSolver** tool for extracting pushdown control-flow graphs from Java programs. This tool produces output in the format used by PDSolver; by combining it with a script for converting PDSolver problems to MCECTL ones, we can obtain a convenient way of applying the model checker to real programs.

Example:

```
package ectl;

public class Files {

   public static void open_handle() { }

   public static void close_handle() { }

   public static void main( String[] args ) {
      open_handle();
   }
}
```

Output from JimpleToPDSolver, filtered through `convert.pl`:

```
PDS jimple_pds {
   STATE ( csend[cpl_ectl_files__open_handle_v__5_5] :
            csend,cpl_ectl_files__open_handle_v__5_5 )
   STATE ( csend[cpl_ectl_files__main_aljava_lang_string_v__11_4] :
         csend,cpl_ectl_files__main_aljava_lang_string_v__11_4 )
   STATE ( csend[cpl_ectl_files__main_aljava_lang_string_v__10_1] :
         csend,cpl_ectl_files__main_aljava_lang_string_v__10_1 )
   STATE ( csend[cpl_ectl_files__main_aljava_lang_string_v__10_0] :
         csend,cpl_ectl_files__main_aljava_lang_string_v__10_0 )
   STATE ( csend[cpl_ectl_files__main_aljava_lang_string_v__10_3] :
         csend,cpl_ectl_files__main_aljava_lang_string_v__10_3 )
   STATE ( csend[cpl_ectl_files__main_aljava_lang_string_v__10_2] :
         csend,cpl_ectl_files__main_aljava_lang_string_v__10_2 )
   STATE ( csend[cplinit006] :
         csend,cplinit006 )
   STATE ( csend[_] :
         csend,_ )
   STATE ( csinit[cpl_ectl_files__open_handle_v__5_5] :
         csinit,cpl_ectl_files__open_handle_v__5_5 )
   STATE ( csinit[cpl_ectl_files__main_aljava_lang_string_v__11_4] :
         csinit,cpl_ectl_files__main_aljava_lang_string_v__11_4 )
   STATE ( csinit[cpl_ectl_files__main_aljava_lang_string_v__10_1] :
         csinit,cpl_ectl_files__main_aljava_lang_string_v__10_1 )
   STATE ( csinit[cpl_ectl_files__main_aljava_lang_string_v__10_0] :
         csinit,cpl_ectl_files__main_aljava_lang_string_v__10_0 )
   STATE ( csinit[cpl_ectl_files__main_aljava_lang_string_v__10_3] :
         csinit,cpl_ectl_files__main_aljava_lang_string_v__10_3 )
   STATE ( csinit[cpl_ectl_files__main_aljava_lang_string_v__10_2] :
         csinit,cpl_ectl_files__main_aljava_lang_string_v__10_2 )
   STATE ( csinit[cplinit006] :
```

[8] http://www.comlab.ox.ac.uk/matthew.hague/pdsolver.html

```
      csinit,cplinit006 )
   STATE ( csinit[_] :
      csinit,_ )
   STATE ( csq[cpl_ectl_files__open_handle_v__5_5] :
      csq,cpl_ectl_files__open_handle_v__5_5 )
   STATE ( csq[cpl_ectl_files__main_aljava_lang_string_v__11_4] :
      csq,cpl_ectl_files__main_aljava_lang_string_v__11_4 )
   STATE ( csq[cpl_ectl_files__main_aljava_lang_string_v__10_1] :
      csq,cpl_ectl_files__main_aljava_lang_string_v__10_1 )
   STATE ( csq[cpl_ectl_files__main_aljava_lang_string_v__10_0] :
      csq,cpl_ectl_files__main_aljava_lang_string_v__10_0 )
   STATE ( csq[cpl_ectl_files__main_aljava_lang_string_v__10_3] :
      csq,cpl_ectl_files__main_aljava_lang_string_v__10_3 )
   STATE ( csq[cpl_ectl_files__main_aljava_lang_string_v__10_2] :
      csq,cpl_ectl_files__main_aljava_lang_string_v__10_2 )
   STATE ( csq[cplinit006] : csq,cplinit006 )
   STATE ( csq[_] : csq,_ )
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__10_2] ->
      csq[REWRITE cpl_ectl_files__main_aljava_lang_string_v__11_4])
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__10_1] ->
      csq[REWRITE cpl_ectl_files__main_aljava_lang_string_v__10_3])
   ACTION ( a : csq[_] ->
      csend[REWRITE _])
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__10_1] ->
      csq[REWRITE cpl_ectl_files__main_aljava_lang_string_v__10_2])
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__11_4] ->
      csq[POP])
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__10_0] ->
      csq[PUSH cpl_ectl_files__main_aljava_lang_string_v__10_1])
   ACTION ( a : csq[cpl_ectl_files__main_aljava_lang_string_v__10_3] ->
      csq[POP])
   ACTION ( a : csq[cpl_ectl_files__open_handle_v__5_5] ->
      csq[POP])
   ACTION ( a : csinit[_] ->
      csq[PUSH _])
   ACTION ( a : csq[cplinit006] ->
      csq[REWRITE cpl_ectl_files__main_aljava_lang_string_v__10_0])
}
```

If we add the following simple check:

```
DFA dfa {
   STATE ( *s1 )
   ACTION( a: s1 -> s1 )
}

FORMULA phi1 {
   E( 1 U[dfa] csend )
}
:check(phi1, jimple_pds)
```

We get the results:

```
Results: {
   <csend,cpl_ectl_files__main_aljava_lang_string_v__11_4>: T     [ <> s(s1,csend) ]
   <csend,cpl_ectl_files__main_aljava_lang_string_v__10_1>: T     [ <> s(s1,csend) ]
   <csend,cpl_ectl_files__main_aljava_lang_string_v__10_0>: T     [ <> s(s1,csend) ]
   <csend,cpl_ectl_files__main_aljava_lang_string_v__10_3>: T     [ <> s(s1,csend) ]
   <csend,cpl_ectl_files__main_aljava_lang_string_v__10_2>: T     [ <> s(s1,csend) ]
   <csend,cplinit006>: T     [ <> s(s1,csend) ]
```

```
   <csend,_>: T     [ <> s(s1,csend) ]
   <csend,cpl_ectl_files__open_handle_v__5_5>: T     [ <> s(s1,csend) ]
   <csinit,cpl_ectl_files__main_aljava_lang_string_v__10_3>: F
   <csinit,_>: T     [ <> s(s1,csinit) --a--> <_> s(s1,csq) --a--> <_> s(s1,csend) ]
   <csinit,cplinit006>: F
   <csinit,cpl_ectl_files__main_aljava_lang_string_v__10_2>: F
   <csinit,cpl_ectl_files__main_aljava_lang_string_v__10_0>: F
   <csinit,cpl_ectl_files__main_aljava_lang_string_v__10_1>: F
   <csinit,cpl_ectl_files__main_aljava_lang_string_v__11_4>: F
   <csinit,cpl_ectl_files__open_handle_v__5_5>: F
   <csq,cpl_ectl_files__open_handle_v__5_5>: F
   <csq,cpl_ectl_files__main_aljava_lang_string_v__11_4>: F
   <csq,cpl_ectl_files__main_aljava_lang_string_v__10_1>: F
   <csq,cpl_ectl_files__main_aljava_lang_string_v__10_0>: F
   <csq,cpl_ectl_files__main_aljava_lang_string_v__10_3>: F
   <csq,cpl_ectl_files__main_aljava_lang_string_v__10_2>: F
   <csq,cplinit006>: F
   <csq,_>: T     [ <> s(s1,csq) --a--> <> s(s1,csend) ]
}
```

which indicate paths to the end state from various points in the program.

7 Conclusions

This project has successfully implemented a system for automated solution of the global model checking problem for CTL[PDA,DPDA]. To the best of my knowledge, it is the first concrete program for doing this.

7.1 Summary

The system includes a robust and usable infrastructure for defining and displaying regular and pushdown automata and systems as well as Extended CTL formulas. This surrounding infrastructure is sufficiently flexible that it could potentially provide a good basis for model checking of pushdown systems more generally.

A number of additional convenience features are provided: regular expressions allow a more intuitive way to specify automata, and a conversion script is available for compatibility with JimpleToPDSolver.

The core model checking procedure is, for a first implementation, efficient – the worst-case complexity for checking a fixed formula is quadratic in the product of the sizes of the system, automata used, and the stack alphabet.

As well as the basic per-state satisfaction results, the algorithm produces witnessing traces when this is relevant. The system has been thoroughly tested and is believed to produce correct results.

7.2 Project Evaluation

On the whole, the project has been a success. The key algorithms have been implemented, thoroughly tested, and shown to work in practice.

While the overall architecture is sound, there are elements of the design which could be improved upon a little. Though the use of ID numbers to represent states and configurations is well-advised, the approach in its current form can be confusing to the uninitiated, and it would benefit from a more formal exposition. A dedicated subsystem for output and logging would also have been helpful during development – for instance, more granular log levels would have made debugging easier, and should probably have been included from the start. There is also some duplicated code in the construction of the various product systems – it would have been more elegant to introduce a greater degree of abstraction to the process of iterating over rules of automata, which would have helped with this. Thankfully, none of these are fundamental problems and with careful use of regression testing it would be straightforward to correct these imperfections.

The choice of an interactive command line as an interface to the system has been justified by the usability improvements this delivers. Since the nature of the logic entails that multiple automata can be involved in a single formula, a persistent environment with named systems and automata is essential to prevent confusion. Similarly, the encapsulation of functionality into separate command objects proved a worthwhile decision – this contributes greatly to the modularity and extensibility of the system as a whole.

7.3 Further work

While the system works well and demonstrates the potential usefulness of Extended CTL as a logic for program verification, there is much scope to improve upon this initial implementation.

7.3.1 Features

There are a number of features which could improve the usability of the system considerably, if added.

Perhaps most obviously, it would be convenient to allow the non-regular portion of a formula or model to be described by a context-free grammar in the input, say in Backus-Naur Form. Currently it is necessary to explicitly enter states and rules for a pushdown automaton, which can be unintuitive.

Similarly, since the algorithm for checking Release formulas only works with DPDAs, it would be helpful for the system to automatically determinise PDAs when necessary.

7.3.2 Optimisation

Work by Lal and Reps [8] offers an alternative algorithm for reachability analysis of pushdown systems; it may be worth investigating whether their tool is able to improve upon the performance of the wpds library in practice.

For systems with very large numbers of variables, it would be sensible to use Binary Decision Diagrams to represent valuations symbolically. This approach has been undertaken successfully by such projects as Bebop [2] and NuSMV [4]; indeed it has been applied to the checking of pushdown systems at least in the case of LTL [5].

7.3.3 Integration

Finally, there is much scope for better integration with other systems, to better allow verification of real programs. It would be possible, for example, to create an Eclipse plug-in for automatically building a pushdown control flow graph from a source file, and perhaps checking a set of standard formulas against it.

Similarly, it is possible to use GCC to create control flow graphs of C++ code; tools for checking these conveniently could be very useful.

8 Acknowledgements

I am very grateful to Stephan Kreutzer for acting as my supervisor for this project, in which capacity he was most helpful.

I would also like to thank Matthew Hague for his help with searching for automata libraries and general advice concerning PDSolver.

References

[1] R. Axelsson, M. Hague, S. Kreutzer, M. Lange, and M. Latte. Extended computation tree logic. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 67–81. Springer, 2010.

[2] T. Ball and S. Rajamani. Bebop: A symbolic model checker for boolean programs. *SPIN Model Checking and Software Verification*, pages 113–130, 2000.

[3] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. *CONCUR'97: Concurrency Theory*, pages 135–150, 1997.

[4] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, pages 241–268. Springer, 2002.

[5] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *Computer Aided Verification*, pages 324–336. Springer, 2001.

[6] Javier Esparza, David Hansel, Peter Rossmanith, and Stefan Schwoon. Efficient algorithms for model checking pushdown systems. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of CAV 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer, July 2000.

[7] M. Hague and CHL Ong. Analysing mu-calculus properties of pushdown systems (tool presentation). *Submitted to SPIN*, 2010.

[8] A. Lal and T. Reps. Improving pushdown system model checking. In *Computer Aided Verification*, pages 343–357. Springer, 2006.

[9] R.S. Mitra. Strategies for mainstream usage of formal verification. In *Proceedings of the 45th annual Design Automation Conference*, pages 800–805. ACM, 2008.

[10] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[11] I. Walukiewicz. Pushdown processes: Games and model checking. In *Computer Aided Verification*, pages 62–74. Springer, 1996.

A User Manual

A.1 Compilation

If you wish to build MCECTL yourself, you will require the following:

```
CMake >= 2.8.2
Boost >= 1.43.0
GNU Bison >= 2.4.3
flex >= 2.5.35
GNU Readline >= 6.1.002
```

Earlier versions may work, but this has not been tested.

All other libraries used are provided with the source distribution. These include:

```
libfa >= 0.7.4 (part of the Augeas project)
wpds >= 16/05/2006 (available from the Institute of Formal Methods
in Computer Science, University of Stuttgart)
```

From the main directory, the following commands will build the system:

```
$ cmake .
$ make
```

This produces two executables: **MCECTL-REPL**, for using the system interactively, and **Test**, which runs the Boost unit tests.

A.2 MCECTL-REPL

MCECTL-REPL takes two command line options.

```
--verbose          If this flag is present, produce more detailed output
--file filename    Load and execute commands from a file
```

Upon running MCECTL-REPL, a prompt will be displayed:

```
MCECTL >
```

at which commands in the MCECTL input language may be entered.

A.3 Input Language

The MCECTL input language is defined as follows. Definitions are signalled by the keyword for the type of data being defined, in all caps (e.g. PDS) followed by the actual definition, in curly braces.

Other commands are in lower case and preceded by a colon. These include, for example, checking a formula against a system.

```
DFA my_dfa {
   STATE(  my_state_1 )
   STATE(  my_state_2 )
   STATE( *my_accepting_state )
   ACTION( a : my_state_1 -> my_accepting_state )
}
Declare a finite automaton explicitly. * indicates a final state.


REGULAR my_regex {
a* b*
}
Declare a finite automaton by providing a regular expression.


PDA my_pda {
   STATE ( empty )
   STATE ( toast_ready )
   STATE ( *fulfilled )
   ACTION ( make_toast: empty[_] -> toast_ready[PUSH toast] )
   ACTION ( eat_toast: toast_ready[toast] -> fulfilled[POP] )
   ACTION ( make_and_eat: toast_ready[toast] -> toast_ready[REWRITE toast] )
}
Declare a pushdown automaton. * indicates a final state.


LTS my_lts {
   STATE ( s1: p )
   STATE ( s2: q )
   STATE ( s3: )
   ACTION ( a: s1 -> s2 )
   ACTION ( b: s2 -> s3 )
}
Declare a labelled transition system.


PDS pds1 {
```

```
STATE( p1[_] : )
STATE( p1[s] : )
STATE( p2[_] : )
STATE( p2[s] : p)
ACTION( a: p1[_] -> p1[PUSH s] )
ACTION( a: p1[s] -> p1[PUSH s] )
ACTION( b: p2[_] -> p2[PUSH s] )
ACTION( c: p1[s] -> p2[POP] )
}
Declare a pushdown system.


FORMULA my_formula {
A( (p & E(!q U[dfa1](r -> p))) U[dfa2] (q|A(0 R[dfa1] !EX r)) )
}
Declare a formula.  Formulas are input as follows:
0                     true
1                     false
p                     proposition
!formula              negation
(f & g)               conjunction
(f | g)               disjunction
(f -> g)              implication
E(f U[automaton] g)   exist until
E(f R[automaton] g)   exist release
A(f U[automaton] g)   all until
A(f R[automaton] g)   all release
EX f                  exist next
AX f                  all next


:load("input.ectl")
Load commands from the specified file. Tab-completion available.


:quit
End the session. (Also ctrl-D)


:check(my_formula, my_system)
Check which states of the system model the formula.
If the system is an LTS, any formula automata should be PDAs.
If the system is a PDS, the formula automata should be DFAs.


:show(my_formula)
:show(my_automaton)
:show(my_system)
```

```
Print a textual description of the named object.


:xshow(my_automaton)
:xshow(my_system)
Display a graphical representation of the named automaton or system.
This requires that GraphViz is installed, and that the 'dot'
tool is present in the system PATH.
```

A.4 Conversion from JimpleToPDSolver output

To use the PDSolver integration script, simply pipe in the output from JimpleToPDSolver:

```
$ tools/JimpleToPDSolver "<ectl.Files: void main(java.lang.String[])>"
                         -pds -f jimple.out

$ scripts/convert.pl < jimple.out > files.ectl
```

This creates a pushdown system from the control flow of the `ectl.Files` Java program, and stores the appropriate MCECTL input in `files.ectl`, ready to be checked.

```
$ ./MCECTL-REPL --file files.ectl
```

B Code Listings

In the following listings, please note that many of the less important classes (e.g. exceptions and AST nodes) have been omitted for brevity. The full source code is available from the author upon request.