Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Miguel Sousa Cancelinha

Towards model checking Electrum specifications with LTSmin

October 2019

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Bruno Miguel Sousa Cancelinha

Towards model checking Electrum specifications with LTSmin

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Professor Doutor Alcino Cunha
Professor Doutor Paulo Sérgio Almeida

October 2019

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

ACKNOWLEDGEMENTS

When writting the acknowledgments of a master thesis, we are not just acknowledging the people who were close by and helped on the construction of the whole dissertation. Rather it is my view that we are acknowledging everyone who accompanied us in these past five years. I am certantily not the same person I was five years ago, and I am sure I wont be the same person five years from now. But that transformation cannot be attributed to a single person or a group, but to everyone I have ever interacted with; in the sense that every person I stumble upon during my life, in some way or another, changes me and the path I am following. I cannot therefore thank individually each one who helped bringing me here, since I would either have inadequately long acknowledgments or I would have committed the terrible disgrace of missing someone.

With that cleared out, I must thank those who, like me, where thrown into a new world five years ago and who have been supporting me ever since. I also send my thanks: To those friendships that begun on Wednesday nights, and prolonged during the years. To those whom culture and the joy of argumentation brought us together. To those that I have recently met through activism. To those with whom I have lived with, and had to put up with me. To everyone I have ever drank tea with. To everyone I have ever loved. To the people at CeSIUM with whom I have learned so much. To the volunteers of CoderDojo and the kids we teach. To the community around Include Braga. To all my professors and teachers. To the cafés that provided the tranquil environment that allowed for this dissertation to be written. To my long-lasting friends in Chaves that sometimes had to assume I went missing. To my family and our pets.

Finally I must also acknowledge, this time by name, those who directly helped in the development of this dissertation. Namely Professor Alcino Cunha for always keeping the ambition and motivation high; Professor Paulo Sérgio Almeida, for always bringing a different perspective on things; Nuno Macedo whom was always ready to help me and discuss my problems; Alfons Laarman for having the patience of answering my long list of questions on LTSmin.

This work is financed by the ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826.

STATEMENT OF INTEGRITY

RESUMO

Model checking é uma técnica comum de verificação; garante a consistência e integridade de qualquer sistema fazendo uma exploração exaustiva de todos os possíveis estados. Devido à grande quantidade de intercalações possíveis entre eventos, modelos de sistemas distribuídos muitas vezes acabam por gerar um número de estados muito grande. Nesta dissertação vamos explorar os efeitos de *partial order reduction* — uma técnica para mitigar os efeitos da explosão de estados — implementando uma linguagem semelhante ao Electrum com LTSmin. Vamos também propor um *event layer* por cima do Electrum e uma análise sintática para extrair informação necessária para que esta técnica possa ser implementada.

Palavras-chave: Alloy, Electrum, Model checking, LTSmin, Partial order reduction, TLA$^+$

ABSTRACT

Model checking is a common verification technique to guarantee the consistency and integrity of any system by an exhaustive exploration of all possible states. Due to the large amount of interleavings, models on distributed systems often end up with a huge state-space. In this dissertation we will explore the effects of partial order reduction — a technique to mitigate the effects of this state-explosion problem — by implementing an electrum-like language with LTSmin. We will also propose an event layer over Electrum and a syntactic analysis to extract valuable information for this technique to be implemented.

Keywords: Alloy, Electrum, Model checking, LTSmin, Partial order reduction, TLA$^+$

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

In a time where computational systems more and more affect our daily lives, the need for fully verified and error free software grows. In critical systems specially, the high demand for reliable software requires mathematical techniques to ensure its correctness. Formal methods consists of multiple mathematical techniques to, even at an early stage of the design process, guarantee the consistency and integrity of any system.

One of the most widely-used formal method techniques is *model-based* verification, which relies on models mathematically describing a system's evolution. Having this model one can, through exausitive exploration – meaning testing every possible scenario –, verify certain specified properties about the system's behaviour. This technique is called *model checking*.

Due to the somewhat recent improvements in computer science with enhanced processing power, and faster algorithms and data structures, model checking techniques have been gaining relevance for verifying real-world systems (Baier and Katoen (2007)). Many model checking tools like Alloy and TLA$^+$are being regularly used by governments and business. TLA$^+$, for instance, is being used by the Amazon team Newcombe (2014).

Today with the proliferation of IoT technologies and decentralized services, modern systems often exhibit some concurrent characteristics. Thus various model checking solutions exists like *Temporal Logic of Actions* (TLA$^+$) (with its associated checker, TLC) and Alloy [1]. TLA$^+$is known for its efficiency although this is achieved by severely limiting the language expressiveness. Alloy, on the other hand, is famous for its unrestrained expressiveness allowing a variety of ways to specify systems and properties.

Electrum is an expansion of Alloy that is being devoleped at the University of Minho in Portugal, and ONERA in France. It aims to get the efficiency of TLA$^+$while mantaining the expressiveness of Alloy. It borrowed concepts from TLA$^+$, like *actions* that model the evolution of the system over time, and allows properties to be defined using *Linear Temporal Logic* (LTL). Although having most of the features associated with TLA$^+$, Electrum still trails behind TLC when compared to its efficiency.

Nevertheless, TLA$^+$inspired features make Electrum appealing for modelling distributed systems. However, the nature of such systems usually implies asynchronous communication

1 Technically, Alloy is a model finder, but in practice is frequently used as a bounded model checker, mainly for safety properties

and independence between actions. This turns out to be a huge burden for model checkers, that end up recreating various system traces that are all identical to each other, modulo re-ordering of independent actions. *Partial order reduction* reduces the state space by removing irrelevant system traces and it is a technique in model checkers, but it is not currently being implemented by Electrum.

In this dissertation we will analyze different techniques to improve Electrum and propose new changes to its language and the Analyzer. We will first take a deep look at how model checking works in Chapter 2, running through the history of model checking and describing a technique to reduce the overall state space. In Chapter 3 we describe the Electrum model checker and learn how to specify and analyze properties for it. Later, in Chapter 4, we study the LTSmin model checker and its usages; we will also take a deep look at its unique algorithm for state space reduction. Finally, in Chapter 5 we will combine what we have learned from the past chapters and present an electrum-like language with LTSmin. We wind up this dissertation with its final conclusions in Chapter 6.

2 MODEL CHECKING

Model checking began in the early 1980s in works such as those developed by Emerson and M. Clarke (1982). It is a way to automatically verify formal specifications in a model of a system, this is achieved by an exhaustive exploration of all possible system traces to check desirable properties.

In this chapter we will take an overview of the theory of model checking and study multiple techniques to automatically verify formulas in a model.

The process of model checking a system usually is deconstructed in three steps: *Modelling*, translate the system to a model acceptable by the model checking tool; *Specification*, formalize the system requirements into properties understandable by the tool; *Verification*, the process of checking that a property holds in the model. In theory, the last step is fully automated by the tool but usually the verification ends up revealing errors, either in the model or in the specification, which must be fixed in order to get trustworthy results. In this section we shall go more in depth in each of these steps and explain the theory behind them.

2.1 MODELLING

For the purposes of the present dissertation, we will mostly consider reactive systems which continuously interact with their environment. Because of their infinite nature, these systems cannot be modeled by simply describing their input-output behaviour; as such we must conceive of another way to abstract how they behave.

A key component to understand this way of looking at systems is the notion of a *state*. A state is a snapshot, a look at the values of all the variables of the system at one instant. Now we can imagine how a system evolves by going from one state to another: a set of *transition* defined as pairs of states. In short, we can perceive the system as a *state machine* and the *computation* as a sequence of states reached by following the transitions. Thus, we get the implicit notion of time and the evolution of the system.

One possible way to represent this state machine is by a *Kripke structure* (Kripke (1963)). We define a Kripke structure M over the atomic propositions AP as the pair $M = \langle S, S_0, R, L \rangle$, where:

1. S is the set of all possible system states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is the relation that defines the transition from one state to the other.
4. $L : S \to 2^{AP}$ is the labelling function that labels each state with the set of atomic propositions from AP true in that state.

Relation R is assumed to be total i.e. for every $s \in S$ there is a $s' \in S$ where $\langle s, s' \rangle \in R$, meaning that for every state s there is a transition to another state s'. We will mostly denote $s\ R\ s'$ instead of $\langle s, s' \rangle \in R$.

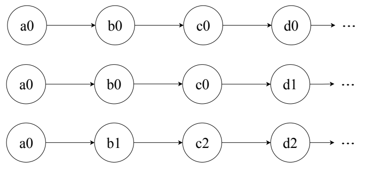
From the Kripke structure we can infer the notion of *paths*. A path — denoted by π — is an infinite sequence of states; $\pi = s_0 s_1 s_2 \ldots$ represents the path starting on state s_0 and where $\forall i \geq 0 : \langle s_i, s_{i+1} \rangle \in R$. We will also use π_i to be the i-th state from path π, π^i to denote the sub-path of π beginning in the ith state, $Paths(s)$ represents all possible paths beginning in state s, and $Paths(M)$ represents all possible paths of model M. These notions will be useful for defining the semantics of our specification logic.

2.2 SPECIFICATION

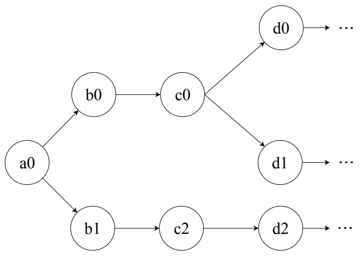
Most of the software requirements are specified in a natural language; however, natural languages are frequently susceptible to ambiguity and cannot be easily translated to logical formulas. Formal logic specifications, on the other hand, allow for a unique interpretation. In general, these specifications fall into two categories: *liveness properties*, usually characterised by *"Eventually something good will happen"*; and *safety properties*, of the kind *"Something bad never happens"*.

Such notions of "eventually" and "always" (negation of "never") refer to the evolution of a system without explicitly referring to time. We can describe these notions in a formal language using *temporal operators*. We have an intuitive understanding of what these terms ("eventually" and "always") mean, however we must define them in the context of Kripke structures as well as show how atomic propositions can be validated in them. This can be achieved by a powerful logic called *temporal logic*.

Temporal logic is an expansion of classical logic to support reasoning over time. In computer science there are two main temporal logic formalisms: *branching time* and *linear time*, which are ilustrated by Figure 1. In branching time (Figure 1b), the semantics is defined on a computation tree unrolled from a initial state of the Kripke structure; while in linear time (Figure 1a), the semantics is defined over the set of computational paths of the Kripke struucture.



(a) Representation of linear time



(b) Representation of branching time

Figure 1: Two representations of the same computation in linear and branching time

2.2.1 *CTL*

Computation Tree Logic (CTL) is a branching time temporal logic. The interpretation of CTL formulas is defined both in terms of states and paths. Because of its branching-time nature which, with a single tree, allows us to reason about multiple computations, properties defined in CTL are capable of expressing notions such as "in *some* computations X happens".

CTL formulas are classified into *state formulas* and *path formulas*. State formulas assert properties over a state and quantify propositions over paths beginning in that state, whereas path formulas describes temporal properties over one path.

CTL state formulas are defined by the following grammar:

$$\Phi ::= \text{ true} \mid \text{false} \mid \text{a} \mid \Phi_1 \vee \Phi_2 \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

Being AP a set of atomic propositions and a $\in AP$, and φ a path formula. Path formulas, on the other hand, are defined by:

$$\varphi \ ::= \ \bigcirc\Phi \mid \Box\Phi \mid \Diamond\Phi \mid \Phi_1 \,\mathcal{U}\, \Phi_2 \mid \Phi_1 \,\mathcal{R}\, \Phi_2$$

Where Φ is a state formula. Omitting $\neg$, $\wedge$ and $\vee$, the above symbols are read as:

$\exists\varphi$ There exists at least one computation path where φ holds.

$\forall\varphi$ φ holds for all computation paths.

$\circ\Phi$ Φ holds in the next state.

$\Box\Phi$ Φ always holds.

$\Diamond\Phi$ Eventually Φ will hold.

$\Phi_1 \,\mathcal{U}\, \Phi_2$ Φ_1 holds until Φ_2.

$\Phi_1 \,\mathcal{R}\, \Phi_2$ Φ_1 releases Φ_2.

Given a model M and a state formula Φ we say Φ holds in M ($M \models \Phi$) if and only if $\forall s \in S_0 \; : \; M, \; s \models \Phi$, that is, Φ holds for every initial sate of M. We define the satisfaction relation ($\models$) by: Let AP be a set of atomic propositions, $p \in AP$, Φ, Φ_1, and Φ_2 be state formulas and φ be a path formula, and $\not\models$ be the negation of $\models$,

$$\begin{aligned}
M, s \models p &\iff p \in L(s) \\
M, s \models \neg\Phi &\iff M, s \not\models \Phi \\
M, s \models \Phi_1 \wedge \Phi_2 &\iff M, s \models \Phi_1 \wedge M, s \models \Phi_2 \\
M, s \models \Phi_1 \vee \Phi_2 &\iff M, s \models \Phi_1 \vee M, s \models \Phi_2 \\
M, s \models \forall\varphi &\iff \forall\pi \in Paths(s) : M, \pi \models \varphi \\
M, s \models \exists\varphi &\iff \exists\pi \in Paths(s) : M, \pi \models \varphi
\end{aligned}$$

We have just defined $\models$ for state formulas, now let us define it for path formulas. With all the above assumptions and let π be a path in M,

$$\begin{aligned}
M, \pi \models \bigcirc\Phi &\iff M, \pi_1 \models \Phi \\
M, \pi \models \Box\Phi &\iff \forall i \geq 0 : M, \pi_i \models \Phi \\
M, \pi \models \Diamond\Phi &\iff \exists i \geq 0 : M, \pi_i \models \Phi \\
M, \pi \models \Phi_1 \,\mathcal{U}\, \Phi_2 &\iff \exists i > 0 : M, \pi_i \models \Phi_2 \wedge \forall 0 \leq j < i : M, \pi_j \models \Phi_1 \\
M, \pi \models \Phi_1 \,\mathcal{R}\, \Phi_2 &\iff \forall i > 0 : M, \pi_i \models \Phi_2 \vee \exists 0 \leq j < i : M, \pi_j \models \Phi_1
\end{aligned}$$

To understand the difference between the path quantifiers, imagine the following two properties:

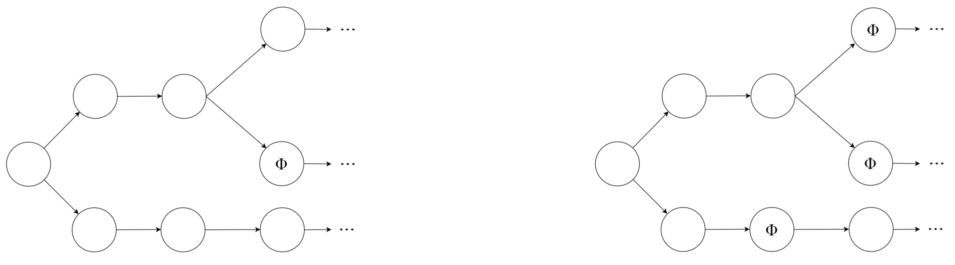
$$\exists\Diamond\Phi \tag{1}$$

$$\forall\Diamond\Phi \tag{2}$$

They are both stating that eventually Φ will hold. However, property 1 is true if at least one of the computational paths satisfies $\Diamond\Phi$, while for property 2 to be true *all* computational paths must satisfy $\Diamond\Phi$. In figure 2a it is represented a model that satisfies property 1 but not property 2, and the model represented in figure 2b satisfies both properties.

2.2.2 *LTL*

Linear Temporal Logic (LTL) is a linear time temporal logic and was actually suggested before CTL by Pnueli (1977). It consists only of path formulas, thus it is unable to define properties



(a) Model that satisfies property (1) (b) Model that satisfies property (2)

Figure 2: The representation of properties (1) and (2) in branching time

involving unrestrained quantification over paths. By definition, LTL formulas apply to all computational paths, as if it was a CTL formula with a single implicit $\forall$.

The syntax of LTL is given by the following grammar:

$$\varphi \ ::= \ true \mid false \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \bigcirc\varphi \mid \Box\varphi \mid \Diamond\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2 \mid \varphi_1 \,\mathcal{R}\, \varphi_2$$

All those symbols are read exactly the same as in CTL, however their semantics change because we do not have a satisfaction relation for state formulas, only for path formulas. So we say that φ holds in model M ($M \models \varphi$) if and only if $\forall \pi \in Paths(M) \cdot M, \pi \models \varphi$. The reader might notice that, although the syntax is very similar, the semantics is not. The satisfaction relation of LTL is defined differently using only path formulas. Let AP be a set of atomic propositions, $p \in AP$, and π^n be a sub-path of π beginning in the n-th state,

$$\begin{aligned}
M, \pi \models p &\iff p \in L(\pi_0) \\
M, \pi \models \neg\varphi &\iff M, \pi \not\models \varphi \\
M, \pi \models \varphi_1 \wedge \varphi_2 &\iff M, \pi \models \varphi_1 \wedge M, \pi \models \varphi_2 \\
M, \pi \models \varphi_1 \vee \varphi_2 &\iff M, \pi \models \varphi_1 \vee M, \pi \models \varphi_2 \\
M, \pi \models \bigcirc\varphi &\iff M, \pi^1 \models \varphi \\
M, \pi \models \Box\varphi &\iff \forall i \geq 0 : M, \pi^i \models \varphi \\
M, \pi \models \Diamond\varphi &\iff \exists i \geq 0 : M, \pi^i \models \varphi \\
M, \pi \models \varphi_1 \,\mathcal{U}\, \varphi_2 &\iff \exists i > 0 : M, \pi^i \models \varphi_2 \wedge \forall 0 \leq j < i : M, \pi^j \models \varphi_1 \\
M, \pi \models \varphi_1 \,\mathcal{R}\, \varphi_2 &\iff \forall i > 0 : M, \pi^i \models \varphi_2 \vee \exists 0 \leq j < i : M, \pi^j \models \varphi_1
\end{aligned}$$

2.3 VERIFICATION

Verification is basically a way to get all states where a formula is valid. Let $M = \langle S, S_0, R, L \rangle$ be a Kripke structure representing a reactive system, and ϕ be a temporal logic formula. The set of states where formula ϕ is valid is denoted by

$$[\![\phi]\!]_M = \{ \, s \in S \mid M, s \models \phi \, \}$$

So, for some formula ϕ to be valid in model M, all the initial states from M must be in $[\![\phi]\!]_M$, that is,

$$M \models \phi \iff S_0 \subseteq [\![\phi]\!]_M$$

In this section we will first look at two ways of verifying CTL formulas by *explicit model chekcing* and *symbolic model checking*. Later we will briefly take a look at how LTL formulas can be verified using Büchi automata, and how LTL formulas can be converted into CTL formulas. Lastly we will look at mechanisms to speedup the verification of both LTL and CTL formulas.

2.3.1 *Model checking for CTL*

Since we wont be using CTL model checking in this dissertation, we will just take a general overview of how model checking is performed in CTL.

Explicit model checking

The first model checking tools verified formulas by transversing an explicit state machine, a technique called *explicit model checking*. Explicit model checking is an approach that checks the validity of some temporal logic formula by explicitly transversing a state machine.

In explicit model checking for CTL, most operators are seen to be redundant and only five basic operators are used ($\neg$, $\vee$, $\exists\bigcirc$, $\exists\,\mathcal{U}$, $\exists\Box$) [1] , and the set of states where some formula is valid is, most of the times pretty direct. For instance, the states $[\![\neg\phi]\!] = S \setminus [\![\phi]\!]$, i.e., the states where some formula $\neg\phi$ is valid is the set of all states minus those where ϕ is valid.

To check $\phi_1 \ \exists\mathcal{U} \ \phi_2$, for instance, we first find $[\![\phi_2]\!]_M$ and then we transverse the graph backwards using the converse of relation R (R°) and register the states from all paths π where $\forall s \in \pi \ : \ s \in [\![\phi_1]\!]_M$.

Checking $\exists\Box\phi$ is more complicated. First, the model is reduced to only those states which do satisfy ϕ, then the state machine is broken down into *nontrivial strongly connected components* (SCC), with the goal of finding a terminal SCC reachable from an initial state, thus trivially ensuring the satisfiability pf $\exists\Box\phi$. This procedure is efficient because a linear

1 For an easier notation, we say $\Phi_1 \ \exists\mathcal{U} \ \Phi_2$ instead of $\exists(\Phi_1 \ \mathcal{U} \ \Phi_2)$

time algorithm to calculate strongly connected components has been proposed by Tarjan (1972).

Symbolic model checking

Symbolic model checking was first proposed by McMillan (1993) who figured out that one could specify a state machine implicitly by using propositional logic with ordered binary decision diagrams (OBDDs).

The first insight is to consider that it is possible to represent the set of formulas can be represented symbolically by a propositional formula. And that most LTL properties can be expressed with either $\exists\Box\phi$ and $\exists\mathcal{U}$, which can be calculated by finding the *fixpoint*.

Given a function f, a fixpoint x is such that $x = f(x)$. In symbolic model checking, a CTL formula is transformed by a function called *predicate transformer* denoted by $\tau : \mathcal{P}(S) \rightarrow \mathcal{P}(S)$. We can then calculate $[\![\phi]\!]$ for any formula ϕ by finding a fixpoint of some τ. For most formulas we need one of two types of fixponts, a *least fixpoint* or a *greater fixpoint*, denoted by μ and v respectively.

$$\begin{aligned} [\![\exists\Box\phi]\!] &= vZ \;\cdot\; \phi \;\wedge\; \exists\bigcirc Z \\ [\![\phi_1 \;\exists\mathcal{U}\; \phi_2]\!] &= \mu Z \;\cdot\; \phi_2 \;\vee\; (\phi_1 \wedge \exists\bigcirc Z) \end{aligned}$$

Having defined $\exists\bigcirc Z$ to be a *quantified boolean formula* (QBF), we can now define the semantics of CTL temporal operators as being the least or the greatest fixpoint of specific predicate transformers. These formulas can then be efficiently calculated using binary decision diagrams.

Although this technique is used to calculate symbolic model checking of CTL formulas, Clarke et al. (1994) demonstrated that LTL model checking can be reduced to CTL model checking under fairness contraints. They have sucessfully translated LTL formulas into SMV models, this ways introducing LTL model checking into SMV.

2.3.2 *Automata-based model checking for LTL*

Vardi and Wolper (1986) proposed a technique to verify LTL properties by using *nondeterministic Büchi automata* (NBA), a structure first suggested by Büchi (1990). The key idea is to translate the entire transition system and the *negation* of the LTL property we want to verify into a Büchi automaton (representing the counter-examples) and then check if there is some intersection between the languages of both automata. This is the technique used by the model checkers SPIN (Holzmann (1997)) and LTSmin (Kant et al. (2015)).

A Büchi automata $\mathcal{A}$ is a finite automata defined by $\mathcal{A} = \langle S, \Sigma, \rho, S_0, F \rangle$, where

1. S is a set of states
2. Σ is an alphabet
3. $\rho : S \times \Sigma \to 2^S$ is a nondeterministic transition function
4. $S_0 \subseteq S$ is the set of initial states
5. $F \subseteq S$ is the set of accepting states

As we stated previously, we are aiming to prove $M \models \phi$ where M is a kripke structure and ϕ an LTL formula. Being $\mathcal{A}_M$ and $\mathcal{A}_{\neg\phi}$ a Büchi automata for the model and for the LTL formula respectively, and $\mathcal{L}(\mathcal{A})$ the language of some Büchi automata. The main challenge becomes:

$$\mathcal{L}(\mathcal{A}_M) \cap \mathcal{L}(\mathcal{A}_{\neg\phi}) = \emptyset$$

Since several standard techniques exist to check the emptiness of an automata, in order to verify an LTL property we just need to know how to calculate the intersection of two automata and how to translate an LTL formula into a Büchi automata.

Intersection of automata

A generalised NBA (GNBA) is similar to a regular NBA but, instead of having a single set of accepting states F, it has a set of sets of accepting states $\mathcal{F}$. It is easy to understand how NBA can be translated into a GNBA: Everything is the same and $\mathcal{F} = \{F\}$.

For two Büchi automata $\mathcal{A}_1 = \langle S_1, \Sigma, \delta_1, S_{0_1}, \mathcal{F}_1 \rangle$ and $\mathcal{A}_2 = \langle S_2, \Sigma, \delta_2, S_{0_2}, \mathcal{F}_2 \rangle$ there is some other Büchi automata $\mathcal{A}$ such that:

$$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$$

This new Büchi automata $\mathcal{A}$ is a GNBA and can be calculated as the cross product of the two auomata, that is:

$$\mathcal{A} = \mathcal{A}_1 \otimes \mathcal{A}_2 = \langle S_1 \times S_2, \Sigma, \delta, S_{0_1} \times S_{0_2}, \mathcal{F} \rangle$$

Where δ is the conjunction of both δ_1 and δ_2, and $\mathcal{F}$ is given by:

$$\mathcal{F} = \{F_1 \times S_2 \mid F_1 \in \mathcal{F}_1\} \cup \{S_1 \times F_2 \mid F_2 \in \mathcal{F}_2\}$$

Translating an LTL formula into a Büchi automaton

First we need to present a couple of definitions on Büchi automata:

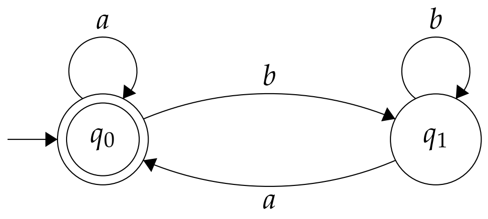
Let *word* v of length $|v|$ over alphabet Σ in the automaton $\mathcal{A}$ denote a possible sequence of transitions of $\mathcal{A}$.

We say that a *run* of $\mathcal{A}$ over the word v is the sequence of states possible by going through each transition of v.

A run is said to be an *accepting* run if the sequence of states ends in some accepting state of F.

Finally, we say that $\mathcal{A}$ *accepts* a word v if and only if there is an accepting run of $\mathcal{A}$ over v.

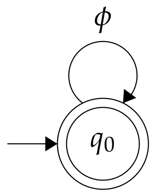
For example, the automaton below – where q_0 is the only accepting state – accepts the empty word (ϵ), *a*, *ba*, *bba*, *bbaa*, *abaa*, et cetera, but not *ab*, for example.



A language of an automata $\mathcal{L}(\mathcal{A})$ is the set of all words accepted by the automata.

All LTL formulas can be represented by a Büchi automaton, we will now look at some examples of this translation. Suppose we have an alphabet $\Sigma = \{\phi, \psi\}$.

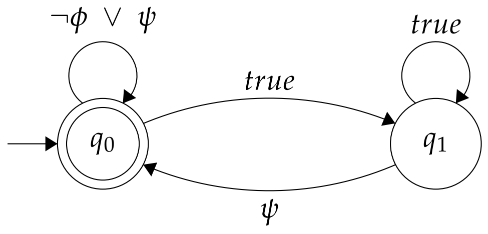
Let us start with a simple example: the LTL property $\Box\phi$ means that ϕ is valid in every state, that is, there is a transition ϕ in every state. We can intuitively consider a language for that formula or at least some words for it. The empty word ϵ is a word in $\mathcal{L}(\mathcal{A}_{\Box\phi})$, as is, ϕ, $\phi\phi$, $\phi\phi\phi$, et cetera. This way, the corresponding Büchi automata for forumla $\Box\phi$ is:



For a more complex property $p_1 = \Box(\phi \Rightarrow \Diamond\psi)$ we have:

$$\begin{aligned} \epsilon &\in \mathcal{L}(p_1) \\ \psi &\in \mathcal{L}(p_1) \\ \neg\phi, \phi, \psi &\in \mathcal{L}(p_1) \\ &\ldots \end{aligned}$$

Therefore we have the following Büchi automaton:



2.3.3 *Bounded model checking for LTL*

In *bounded model checking*, we aim to find paths of lenght k which break a given propositional formula. If no path of size k is found, then the search continues for paths larger than k. We can reduce this into a satisfiability formula with propositional logic and pass it to a SAT solver.

Bounded model checking was introduced by Biere et al. (1999) and presented as a fast way to calculate minimal length counterexamples, and one that uses less space when compared to approaches based on binary decision diagrams as is the case with symbolic model checking. In this subsection we will study how it is possible to translate LTL formulas into propositional logic formulas that can be checked by a SAT solver.

Semantics

Although the aim is to get a finite k-sized prefix of a (possibly infinite) path, such prefix can still represent an infinite path if there is a *back loop*, that is, if the successor of the last state is another state inside the prefix. This little nuance makes huge differences when looking at the semantics for LTL. Take a look at the formula $\Box\phi$ which states that "ϕ will always be true", in a non-looping finite path of size k such formula cannot be truthfully verified since ϕ might not hold for some state after k. When defining the semantics for bounded model checking we must be mindful of whether we are talking about looping or non-looping paths.

In non-looping paths we consider that $\Box\phi$ is always false. The semantics for $\phi\ \mathcal{R}\ \psi$ also changes as we have to exclude the possibility where ϕ always holds and ψ is not in the bounded path. This effectively eliminates the duality between $\Box$ and $\Diamond$ since we can no longer say that $\Box\neg\phi \equiv \neg\Diamond\phi$, and between $\mathcal{R}$ and $\mathcal{U}$ since $\neg(\phi\ \mathcal{U}\ \psi) \not\equiv (\neg\phi)\ \mathcal{R}\ (\neg\psi)$.

Translation

Now we will take a look at the problem of reducing an LTL formula into a CNF formula understandable by a SAT solver. We will represent the finite sequences of states $s_0, s_1, \ldots, s_k$ composing the k-sized path that satisfy an LTL formula ϕ in a Kripke model M by $[\![M, \phi]\!]_k$.

To define $[\![M,\phi]\!]_k$, we must first define $[\![M]\!]_k$ that represents all possible k-sized paths in M and then we restrict that sequence to those paths in $[\![\phi]\!]_k$ We define $[\![M]\!]_k$ formally as follows: Let $M = \langle S, S_0, R, L\rangle$ be a Kripke structure, $s_i \in S$ and $s_0 \in S_0$,

$$[\![M]\!]_k = I(s_0) \wedge \bigwedge_{i=0}^{k-1} \langle s_i, s_{i+1}\rangle \in R$$

Where I represents the condition for the initial state.

Now to calculate $[\![\phi]\!]_k$ we must be careful to distinguish from non-looping and looping paths because of the two different semantics. We assume that ϕ was converted into negation normal form. Let us start by expanding $[\![\phi]\!]_k$ to $[\![\phi]\!]_k^i$ where i is the current position in the k-sized prefix, and define it for **non-looping** paths:

$$\begin{aligned}
[\![p]\!]_k^i &= p_i \\
[\![\neg p]\!]_k^i &= \neg p(s_i) \\
[\![\phi \wedge \psi]\!]_k^i &= [\![\phi]\!]_k^i \wedge [\![\psi]\!]_k^i \\
[\![\phi \vee \psi]\!]_k^i &= [\![\phi]\!]_k^i \vee [\![\psi]\!]_k^i \\
[\![\Box\phi]\!]_k^i &= \emptyset \\
[\![\Diamond\phi]\!]_k^i &= \vee_{j=i}^{k} [\![\phi]\!]_k^j \\
[\![\circ\phi]\!]_k^i &= \ if\ i < k\ then\ [\![\phi]\!]_k^{i+1}\ else\ \emptyset \\
[\![\phi\ \mathcal{U}\ \psi]\!]_k^i &= \vee_{j=i}^{k}([\![\psi]\!]_k^j \wedge \wedge_{n=i}^{j-1}[\![\phi]\!]_k^n) \\
[\![\phi\ \mathcal{R}\ \psi]\!]_k^i &= \vee_{j=i}^{k}([\![\phi]\!]_k^j \wedge \wedge_{n=i}^{j}[\![\psi]\!]_k^n)
\end{aligned}$$

When dealing with **looping paths**, the notation $[\![\phi]\!]_k^i$ is not sufficient so we expand it to $_l[\![\phi]\!]_k^i$ where l represents where the loop starts. Let ϕ and ψ be two LTL formulas, $k, i, j \in \mathbb{N}$ and $l, i \leq k$, $succ(s_i)$ be the successor state of s_i such that $succ(i) = i + 1$ and $succ(k) = l$

$$
\begin{aligned}
{}_l[\![\phi]\!]_k^i &= \phi(s_i) \\
{}_l[\![\neg\phi]\!]_k^i &= \neg\phi(s_i) \\
{}_l[\![\phi \wedge \psi]\!]_k^i &= {}_l[\![\phi]\!]_k^i \wedge {}_l[\![\psi]\!]_k^i \\
{}_l[\![\phi \vee \psi]\!]_k^i &= {}_l[\![\phi]\!]_k^i \vee {}_l[\![\psi]\!]_k^i \\
{}_l[\![\Box\phi]\!]_k^i &= \wedge_{j=min(i,l)}^{k} {}_l[\![\phi]\!]_k^j \\
{}_l[\![\Diamond\phi]\!]_k^i &= \vee_{j=min(i,l)}^{k} {}_l[\![\phi]\!]_k^j \\
{}_l[\![\circ\phi]\!]_k^i &= {}_l[\![\phi]\!]_k^{succ(i)} \\
{}_l[\![\phi \,\mathcal{U}\, \psi]\!]_k^i &= \vee_{j=i}^{k}({}_l[\![\psi]\!]_k^j \wedge \wedge_{n=i}^{j-1} {}_l[\![\phi]\!]_k^n) \vee \\
&\quad \vee_{j=l}^{i-1}\,({}_l[\![\psi]\!]_k^j \wedge \wedge_{n=i}^{k} {}_l[\![\phi]\!]_k^n \wedge \wedge_{n=l}^{j-1} {}_l[\![\phi]\!]_k^n) \\
{}_l[\![\phi \,\mathcal{R}\, \psi]\!]_k^i &= \wedge_{j=min(i,l)}^{k} {}_l[\![\psi]\!]_k^j \vee \\
&\quad \vee_{j=i}^{k}\,({}_l[\![\phi]\!]_k^j \wedge \wedge_{n=i}^{j} {}_l[\![\psi]\!]_k^n) \vee \\
&\quad \vee_{j=l}^{i-1}\,({}_l[\![\phi]\!]_k^i \wedge \wedge_{n=i}^{k} {}_l[\![\psi]\!]_k^n \wedge \wedge_{n=l}^{j} {}_l[\![\psi]\!]_k^n)
\end{aligned}
$$

Now it is necessary to merge these definitions of $[\![\phi]\!]_k$ and ${}_l[\![\phi]\!]_k$ with $[\![M]\!]_k$. Be aware that there are two definitions for formulas depending on whether or not the path is looping. In order to make our translation as general as possible, we first need to find a predicate to distinguish looping paths from non-looping ones. For that, we define the loop condition as: for $k, l \in \mathbb{N}$, let:

$$
{}_lL_k = \langle s_k, s_i \rangle \in R
$$

$$
L_k = \vee_{l=0}^{k} {}_lL_k
$$

Now we can define our general translation to encompass both non-looping paths ($\neg L_k$) and looping ones as

$$
[\![M, \phi]\!] = [\![M]\!]_k \wedge ((\neg L_k \wedge [\![\phi]\!]_k^0) \vee \bigvee_{l=0}^{k} ({}_lL_k \wedge {}_l[\![\phi]\!]_k^0))
$$

2.4 PARTIAL ORDER REDUCTION

We have been describing these methods as if the complete state graph was needed to validate any property. This is, however, not necessary in some cases. And thankfully so, as some systems might end up with a huge number of states to validate, taking up a lot of memory and time, a phenomenon called *state explosion problem*. Frequently in distributed systems, only a subset of the entire state graph needs to be considered. To illustrate this property, imagine a distributed system with n processes running simultaneously each changing their internal variable once, and we want to check that eventually they all change their variable. In such a system, the final result is independent of the ordering in which these actions take place. It is, therefore, only necessary to check a single ordering and not all $n!$ possible ones. This is the aim of the *partial order reduction* technique: to reduce the number of possible scenarios, and as a result, the number of states considered to validate a given formula, thus solving the state explosion problem. It must be noted that this reduction is possible as long as the intermediate states are not necessary to the property being checked.

This is a rather simplistic example, in a more complex system where processes do communicate and not all actions are independent from all the others, the partial order reduction technique must single out those actions which are independent of their ordering and can thus be reduced.

There are many algorithms for applying partial order reduction, but all crucially rely on the notion of *independence* between actions, as Baier and Katoen (2007) point out. Usually, two actions are considered independent when: they do not disable each other and commute, meaning that, the resulting state is the same regardless of their ordering. This notion of independence is often determined by a syntactic analysis, and is then used when selecting which actions to explore. For each state, only a subset of all possible actions is considered. These subsets are sometimes called *stubborn sets*, *ample sets* or *persistent set*, although they are somewhat equivalent as is noted by Geldenhuys et al. (2009).

Partial order reduction techniques have been implemented in symbolic model checking by Alur et al. (1997), although significantly more work has been made on partial order reduction for explicit model checking. Regarding explicit state analysis, there are two main approaches: dynamic and static partial order reduction.

In dynamic partial order reduction the reduction is conducted while the property is being checked; Flanagan and Godefroid (2005) demonstrated how a dynamic partial order reduction algorithm can be applied to *software model checking*. In software model checking, instead of having an abstract model to check, we have a software program which gives us access to program counters and other features that facilitate the detection of independent actions by static analysis. More recently, Aronis et al. (2018) showed how dynamic partial order reduction can be applied using *wakeup trees* and an important *happens-before relation*. Although both these approaches were capable of detecting dead-lock and check safety properties, they both relied on the notion of process and were incapable of checking arbitrary LTL properties.

As for static partial order reduction, in contrast with dynamic POR, the state graph is first reduced and only then the analysis is performed. Flanagan and Godefroid (2005) presented a static partial order reduction algorithm capable of both safety and liveness property checking, although still adhering to a process-centred mindset. This technique is currently being used by the SPIN model checker. Finally, Laarman et al. (2016) were able to develop a process-agnostic technique, based on the stubborn sets of Valmari (1991), suited for checking both safety and liveness properties. This last algorithm is currently being used by the LTSmin model checker. We will aim at introducing partial order reduction on Electrum which, as it will be clearer on the following chapter, does not have a notion of process. Therefore, due to its highly flexible characteristics and its indifference towards processes, we will study this algorithm in more detail in section 4.3 in the chapter dedicated to LTSmin.

3 ELECTRUM

Model checking is a verification technique that explores every possible system behaviour, and in which previously specified properties over the system are verified. Alloy, introduced by Jackson (2006), is one of the most popular model checking frameworks. Together with a quite flexible specification language, Alloy's Analyzer adopts *relational logic* — an expansion for *first-order-logic* with *transitive closures* — as its main logical formalisms. Electrum (Macedo et al. (2016)) expands these formalisms to include *temporal logic operators* without limiting Alloy's flexibility.

In this chapter we will take a look at the Electrum's language and its model checker.

3.1 LANGUAGE

Model checking frameworks usually have two distinct languages: one language for describing systems, and another one for specifying properties over said systems. Indeed, these two use cases are widely different. However, in Alloy — and therefore Electrum — these two languages are not separated, instead having a single language empowered with propositional and relational logic for both modeling and specifying the system and its properties. This was inspired by TLA^{+}, a formalism that also has a single language to describe both the model and the specifications. As such, in this section we will discuss the language as a whole, explaining first how to model the system and then how to write properties for it.

We will first begin by presenting the system we will model — how the system evolves through time and what properties we should check — later we will learn how we can describe its evolution and its properties in Electrum.

3.1.1 *Showcase example - Poolboy*

Erlang is an functional programming language based on actors. An actor is a lightweight entity in a separate virtual machine that communicates with other actors through asynchronous message passing. An actor reacts to the different messages it receives in its mailbox.

Poolboy is an Erlang library that allows a user to checkout an actor (called worker), use it in whatever way it needs, and check it back in. It allows for fast attribution of workers by preemptively spawning them and hand them down when needed. It is possible, however, for the client issuing the checkout to timeout, in which case it should not receive the requested worker.

Poolboly is configured by a file that sets the number of initial workers (*iw*) and the maximum overflow (*mo*) of workers. When Poolboy has given all its initial workers, it spawns up to *mo* more. This means that, at maximum, Poolboy is capable of giving $iw + mo$ number of workers.

We chose to model poolboy because of its easy to understand behaviour and due to the fact that the older versions where found to have some bugs by Thompson (2012).

From this example, we can identify the following requirements:

1. There should never be more workers that those allowed by the configuration file.
2. If a client sends a checkout request when there are available workers, provided it doesn't timeout, it will receive a worker.
3. If a client times out and doesn't request another worker, it will not receive a worker.

3.1.2 *Modeling the system*

The full Poolboy specification in Electrum can be found at Appendix A. For now, we will take a look at the most important parts of the model in order to better understand how an Electrum specification of this kind is implemented.

Actors

We will begin by defining the actors that will interact with Poolboy. The prime way of defining these objects is to use *signatures*; a signature defines a set of *atoms*. In Alloy atoms are *indivisible* and *immutable* primitive entities without any built-in properties.

Poolboy has three main actors: Poolboy itself, clients that communicate with Poolboy, and the workers. To our model, it is not relevant whether or not Poolboy is an actor, therefore we will only specify clients and workers.

```
abstract sig Actor {}

sig Worker extends Actor {}
sig Client extends Actor {}
```

Since signatures are sets, it might be helpful to interpret the above lines through the lens of set theory. An *abstract signature* has no elements besides those that belong to its extensions, so we could describe set *Actor* as:

$$Actor = Worker \cup Client$$

The **extends** keyword forces the sets *Worker* and *Client* to be disjoint although both contained in the set *Actor*.

$$Worker \cup Client \subseteq Actor \wedge Worker \cap Client = \emptyset$$

According to the system we are aiming to model, a Client can have a set of workers; we should therefore relate the two. A *relation* is a set of tuples that relates atoms of two or more sets. One can easily expand signatures to include declarations of relations as follows:

```
sig Client extends Actor {
  var workers : set Worker
}
```

This introduces the relation `workers` whose domain is *Client* and range is a set of *Worker*. In other words, it relates every client with its set of workers. An example of this set written using common mathematical notation is something like:

$$workers = \{\langle c_1, w_1\rangle, \langle c_1, w_2\rangle, \langle c_2, w_3\rangle\}$$

In the above example, client c_1 has two workers (w_1 and w_2) while client c_2 only has one (w_3).

The `Actor` previously defined is also a relation, since it is a set of *unary* pairs; these types of relations are usually called *unary relations*. In fact, every set is a unary relation.

Notice that the relation `workers` is declared with **var**, designating it as a *variable* relation, i.e. it can change over time. This allows clients to check in and out workers over time. Variable sets and relations are part of Electrum, which contrasts heavily with the fully static nature of Alloy.

The actors in our system can be alive, dead or blocked; an actor can only be blocked if it is alive. Through out the execution — miming the system we are modeling — clients can be killed and revived and can be blocked and unblocked. We will add these notions to our model by declaring new subsets of actors:

```
var sig alive in Actor {}
var sig blocked in alive {}
```

First, we declare these signatures as variable, since they change over time. Any actor, be they Client or Worker, can die and revive. Now remember the usage of the keyword **extends** we mentioned previously; in the above specification, we use the keyword **`in`** instead. **`extends`** means that all other sets expanding the same superset are disjoint, however **`in`** is much more loose and means, in the above declaration of `alive`, that any Actor can be *in* the subset alive. Only actors that are alive can be blocked, so we keep `blocked` as a subset of alive. The keyword **`in`** should not be confused with the set membership operator (∈) of set theory, in fact it more appropriately represents the operator *subset* (⊆).

Having all our actors defined, let us now move on to Poolboy itself.

```
abstract sig Message {}

one sig MaxOverflow in Int {}

one sig Poolboy {
  var free : set Actor,        -- Every free actor poolboy has
  var overflow : one Int,      -- The number of overflow workers poolboy gave
  var waiting : set Client,    -- Clients waiting for workers
  var mailbox : set Message,   -- Mailbox where Poolboy recives messages
  size : one Int               -- The initial size of free
}
```

Poolboy is defined as **`one`** signature; this is the *multiplicity* keyword. In this case, since `Poolboy` is a unary relation (i.e. just a set), it just represents its size. In Alloy and Electrum the multiplicity of a relation can be:

`no`	An empty set
`one`	A set with only one element
`lone`	A set with either one element or none
`some`	A set with one or more elements

Considering that our system only has one Poolboy, the set representing it must also limit its multiplicity to one. The other relations that are defined in **`sig`** `Poolboy` also have diverse multiplicities. But note that when the relation's range is defined as **`set`** it has no size restrictions.

Any given Erlang actor has a `mailbox`. However, we have limited this to just Poolboy itself, because the client's mailboxes will be abstracted in our model. Otherwise, we could have relation `mailbox` declared inside the signature `Actor`.

Messaging system

In the above code snippet we not only declared **`sig`** `Poolboy` but also another signature `Message`. There is nothing new on that declaration so we can move on to define every message. There are a lot of messages in this system, so it would be unwise to explain all of them in detail in the present document. We will use the definition of `Checkin` as an example.

```
sig Checkin extends Message {
  client : Client,  -- Client issuing the checkin
  actor : Actor     -- Checkin actor
}
```

Remember that Checkin is the act of returning a worker to Poolboy, therefore the message has the reference to the client issuing the checkin and the worker that is being checked in. In the above example we define the message `Checkin` with two relations: `client`, the client issuing the checkin; and `actor`, the worker being returned. By default, the multiplicity of relations declared inside signatures is **one**.

Notice that we did not explicitly force a worker to be checked in — choosing instead to have any kind of actor —, this is because some of the bugs found by Thompson were caused by the client sending an invalid worker.

Now that we have the messages defined, we shall move on to specifying how these messages are sent and received.

```
pred sendMessage[m : Message] {
  m not in Poolboy.mailbox
  mailbox' = mailbox + Poolboy->m
}

pred readMessage[m : Message] {
  m in Poolboy.mailbox
  mailbox' = mailbox - Poolboy->m
}
```

A *predicate* is a boolean-valued function, meaning it can only be either true or false. `sendMessage` has one parameter, `m` of type `message`. `m` is actually a set of multiplicity **one**.

The *dot join* (.) is actually a very important operator, it represents the *composition of relations*. It is not the same as the mathematical composition of functions, nevertheless I will borrow its symbol ($\circ$) to define it in a formal language: Let $\langle a_1, a_2, ..., a_n \rangle \in R$ represent a *n-ary pair* which belongs to the relation R,

$$R \circ S = \{\langle a_1, a_2, \ldots, a_{n-1}, a_{n+1}, \ldots, a_{n+m}\rangle \mid \langle a_1, a_2, \ldots, a_n\rangle \in R \wedge \langle a_n, a_{n+1}, \ldots, a_{n+m}\rangle \in S\}$$

As you can see, to join *two pairs* we check if the last atom of the first pair is equal to the first atom of the second pair; if they are, then the new pair starts with the atoms of the first and ends with those from the second omitting the matched atom. When joining (or composing) *two relations* we join every matching pair.

In line 2 in the declaration of **pred** `sendMessage`, we compose `Poolboy` and `mailbox`. Remember that `Poolboy` is a unary relation with only one element and `mailbox` is a relation between the only atom from `Poolboy` and a set of messages, so by composing the two we get only the set of messages. One can read line 2 quite naturally: *message m is not in Poolboy's mailbox*. We already figured out that `Poolboy.mailbox` is the set of messages received by Poolboy, and because every thing is a set in Alloy/Electrum, `m` is also a set. Remember, **not in** represents the mathematical symbol $\not\subseteq$.

The apostrophe following `mailbox` means: the value of `mailbox` in the immediately following state instance. Please remember that `mailbox` relates `Poolboy` with a set of messsages.

In Alloy and Electrum, the pair $\langle a, b\rangle$ is denoted by `a→b`, however, since everything is a set, the $\rightarrow$ operator more accurately represents the mathematical *cartesian product* ($\times$). So, when in Alloy we write `A→B` being `A` and `B` both sets, we are saying:

$$A \times B = \{\langle a, b\rangle : a \in A \land b \in B\}$$

In line 3, because `Poolboy` and `m` are both sets with multiplicity one, the result of `Poolboy→m` is also a set of multiplicity one with a pair relating the only atom of `Poolboy` and `m`.

Finally, Alloy includes the typical set operators:

- **+** The union of two sets ($\cup$)
- **&** The intersection of two sets ($\cap$)
- **-** The difference of two sets ($\setminus$)

Since `mailbox` is a set of pairs, by adding this new pair (`Poolboy→m`) we are, in effect, adding just a new message. With all that in mind, we can now read line 3 as *the next instance of mailbox is equal to the current instance with a new message*.

This is not imperative programming `sendMessage` does not actually send any message, rather it is more helpful to think of it as a predicate that is true *if and only if* a given message `m` is not currently in `Poolboy.mailbox` but will be in the next state.

Events

Now, if we try to run our system nothing meaningful will happen, the initial state will be generated at random and the variable sets will change chaotically. This is because we have just yet specified *what* objects our system has, now we have to define *how* they will behave.

Firstly we begin by defining the initial state:

```
fact init {
  Poolboy.overflow = 0
  gte[MaxOverflow, 0]
  Poolboy.free = alive & Worker
  Poolboy.size = #Poolboy.free
  no waiting
  no workers
  no blocked
  no mailbox
}
```

A **fact** limits all possible generated traces to only those that satisfy a given formula. If the formula does not contain any temporal operator — as is the case above —, then it only applies to the initial state. This is a pretty straight forward declaration, the only two new elements from the above listing are: `gte` which is a predicate that asserts `MaxOverfow` to be *greater than or equal to* `0`; and `#Poolboy.free` read as *the cardinality of the set Poolboy.free*, that is the number of elements in the set `Poolboy.free`.

Having our initial state specified, we can now focus on defining what can happen in the system. Because there are a lot of actions that can happen, we will just show one as an example.

```
1  pred clientSendCheckin[cli : Client, act : Actor] {
2    cli in (Client & alive) - blocked
3    act in cli.workers
4
5    some msg : Checkin {
6      msg.client = cli
7      msg.actor = act
8      sendMessage[msg]
9    }
10   workers' = workers - cli→act
11
12   alive' = alive
13   blocked' = blocked
14   free' = free
15   overflow' =  overflow
16   waiting' = waiting
17 }
```

The predicate above is divided in three parts: firstly we restrict the given parameters, then we specify what should change and finally we specify what should not change, this last part is called the *frame condition*.

We begin by specifying which Clients and Actors can be the parameters of this action, in this case we say that `cli` can only be an alive client which is not blocked, and `act` is one of `cli`'s workers. Then we move to specify what changes, in this case a `Checkin` message should be sent and `act` should be removed from `cli`'s workers. Here, at line 5, the keyword

some represents an existential quantification over `msg`, in first-order logic using standard mathematical notation this line would translate to,

$$\exists msg \in Checkin : \ldots$$

Universal quantification ($\forall$) is also possible by using the keyword **all**. Lastly, we define the frame conditions, i.e. what doesn't change in this action, by going through every variable relation and force it to stay static.

Again, this is **not** imperative programming. `clientSendCheckin` does not make any changes in the system per se, more like the other way around, `clientSendCheckin` is only true, if and only if the conjunction of all those predicates that compose it is true.

Having specified every other action, we now have to limit the traces to only those that represent meaningful actions by defining a new fact.

```
fact traces {
  always (
    some c : Client | ClientSendCheckout[c] or
    some c : Client, w : Worker | ClientSendCheckin[c, w] or
    some c : Client | ClientSendTimeout[c] or
    PbCheckoutReady or
    PbCheckoutFull or
    ...
  )
}
```

The above code snippet is actually quite interesting, it is the first time we encounter a *linear temporal logic* operator: **always** which is often denoted by $\Box$. If we did not include the **always** operator, we would only be limiting the first state and allow our model to behave chaotically after that.

The above predicate `traces` is of shape $\Box A$, where A is the disjunction of every possible action. This restricts the trace to behave in a specific ways.

We have just described how the system will behave by defining a few predicates, let us call this the *predicate idiom* to specify events. However Electrum is quite versatile and this is but one possible way of defining the system evolution. This approach has its problems: to begin with it is difficult to examine the counter-example traces, since we only see the side-effects and have to extrapolate what event has happened between two consecutive states — which can be nearly impossible if two concurrent events happen at the same time.

We must model our system in such a way that it is clear what is happening. As such, we introduce a new signature that represents the current event (or events) that is (resp. are) taking place. Let us call it *signature idiom* to specify events.

```
one var abstract sig Event {}
```

We define the abstract signature `Event` so that every event that can happen in the system extends it. We made `Event` a **`one`** arity set so that only one event takes place at a time, where it to be a **`no`** arity set, the system would be static and no event possible.

To better understand this new approach, we will encode the event `clientSendCheckin` using a *signature*:

```
var sig ClientSendCheckin extends Event {
  var cli : Client,
  var act : Actor
}
```

Right now, signature `ClientSendCheckin` shows the actors involved in the event — in this case a client and an actor — but does not define what the event does, to specify that we add the following fact:

```
fact {
  all event : ClientSendCheckin {
    event.cli in (Client & alive) - blocked
    event.act in event.cli.workers

    some msg : Checkin {
      msg.c = event.cli
      msg.a = event.act
      sendMessage[msg]
    }
    workers' = workers - event.cli→event.act
  }
}
```

Facts that apply to all atoms of a single signature are best written as *signature facts*, so that referencing the signature is implicit. We expand our declaration of signature `ClientSendCheckin` to include signature facts.

```
var sig ClientSendCheckin extends Event {
  var cli : Client,
  var act : Actor
} {
  cli in (Client & alive) - blocked
  act in cli.workers

  some msg : Checkin {
    msg.c = cli
    msg.a = act
    sendMessage[msg]
  }
  workers' = workers - cli→act
}
```

Notice how we now have no need to write **event**.cli since cli implicitly refers to the relation defined in the signature.

We have no need to explicitly state the frame condition, this is due to the way we will specify what traces are possible. With the predicate idiom we would write the fact trace and explicitly state what actions can occur, in this approach we look for what variables change and imply that some event took place. For example: the only events that change overflow are PbCheckoutReady, PbCheckinOverflow and PbExitOverflow, therefore if overflow changed, one of these events must have taken place. In Electrum we write it the following way:

```
fact updated {
  always (
    (blocked' != blocked
      implies some ClientSendCheckout + ClientSendTimeout + ActorExit +
        PbCheckoutReady + PbCheckoutOverflow + PbCheckinWaiting + PbExitWaiting) and

    (workers' != workers
      implies some ClientSendCheckin + ActorExit + PbCheckoutReady +
        PbCheckoutOverflow + PbCheckinWaiting + PbExitWaiting) and

    (alive' != alive
      implies some ActorExit + PbCheckoutOverflow + PbCheckinOverflow +
        PbExitWaiting + PbExitReady) and

    (free' != free
      implies some PbCheckoutReady + PbCheckinReady +
        PbExitOverflow + PbExitReady) and

    (overflow' != overflow
      implies some PbCheckoutOverflow + PbCheckinOverflow + PbExitOverflow) and

    (waiting' != waiting
      implies some PbCheckoutFull + PbTimeout + PbCheckinWaiting + PbExitWaiting) and

    (mailbox' != mailbox
      implies some ClientSendCheckout + ClientSendCheckin + ClientSendTimeout +
        ActorExit + PbCheckoutReady + PbCheckoutOverflow + PbCheckoutFull +
        PbTimeout  + PbCheckinWaiting + PbCheckinOverflow + PbCheckinReady +
        PbExitWaiting + PbExitOverflow + PbExitReady)
  )
}
```

3.1.3 *Specifying the system*

We have just went through the process of modeling the system. Now we will go through each of the requirements we identified in subsection 3.1.1 and specify properties in Electrum for each of them.

Formalization

Let us recap the requirements we identified previously:

1. There should never be more workers that those allowed by the configuration file.
2. If a client receives a worker, it must be blocked.
3. If a client sends a checkout request when there are available workers, provided it doesn't timeout, it will receive a worker.

We now have to translate this properties from English to temporal relational logic, the specification logic of Electrum.

1. There should never be more workers that those allowed by the configuration file.

This is a typical safety property where we are expecting something bad to never happen, in this case we expect that there should never be more workers than those allowed. The configuration file has the number of workers ($size$) and the maximum overflow ($maxOverflow$) — i.e. how many more workers can Poolboy give —, so the total number of workers allowed by the configuration file is $size + maxOverflow$. Let us assume that $\#totalWorkers$ represents the total number of workers in the system; then in LTL the property is,

$$\Box(\#totalWorkers \leq size + maxOverflow)$$

And reads *always the total number of workers is less or equal to the initial size plus the maximum overflow*. The next requirement is a bit more complex,

2. If a client receives a worker, it must be blocked.

This is a *safety* property, because we are expecting that the proposition remains true through the execution of the system. Let $Rw(c)$ be true if and only if client c receives a worker and $B(c)$ is true if and only if client c is blocked.

$$\Box(\forall c \in Client : Rw(c) \implies B(c))$$

As you can see it becomes quite simple. We now must translate all these specifications from pure LTL to Electrum.

3. *If a client sends a checkout request when there are available workers, provided it doesn't timeout, it will receive a worker.*

This is a *liveness property* because we are expecting something good to eventually happen; in this case, the client will eventually get a worker. However, the client has to first send a checkout request and there must be available workers and the client must not timeout. TLA+ actually has a nice symbol to represent these types of properties, they call it *leads to* ($\leadsto$), but it is only "syntactic sugar" and does not add anything new to LTL because $A \leadsto B$ is defined as $\square(A \Rightarrow \lozenge B)$ (Lamport (2003)). With that said, we will use the purely LTL syntax; any discerning reader will notice the similarities:

$$\forall c \in Client : \square(Co(c) \land overflow < maxOverflow \land \neg T(c) \implies \lozenge Rw(c))$$

Where $Co(c)$ is true if and only if client c sends a checkout message, $T(c)$ is true if and only if client c times out, and $Rw(c)$ is true if and only if client c received a worker.

Property specification

We will now go through each of the requirements we formalize and translate them in Electrum. The first one is quite simple:

$$\square(\#totalWorkers \leq size + maxOverflow)$$

Before we start specifying the property, we need to get the total workers in the system, for that we will use a *function* (**`fun`**). A function is basically a way of reusing the same expression. Every function must specify what it returns and can have multiple arguments.

```
fun total_workers : set Worker {
  Poolboy.free + Client.workers
}
```

In the above example, our function has no arguments and returns a set of `Worker` with all the workers being used in the system. When we say *all workers in the system* we mean those workers that are available in `Poolboy` and those that are being used by some `Client`, so the total set is the union of these two.

Now we can move on to specify the property in Electrum. We will be using an assertion **`assert`** which is a predicate that we can later check.

```
assert maximum_workers {
  always lte[#total_workers, add[Poolboy.size, MaxOverflow]]
}
```

We have already introduce the keyword **`always`** as being the Electrum keyword for $\square$, so let us move to the next specification:

$$\Box(\forall c \in Client : Rw(c) \implies B(c))$$

Let us first define what receive worker (Rw) actually means: we know that a client has received a worker if the cardinality of its set of workers has increased by one, so that is exactly what we specify in Electrum:

```
pred receive_worker[c : Client] {
  #c.workers' = add[#c.workers, 1]
}
```

A client `c` is only blocked if and only if `c in blocked`:

```
assert timeout_is_safe {
  always (
    all c1 : Client {
      receive_worker[c1] implies c1 in blocked
    }
  )
}
```

$$\forall c \in Client : \Box(Co(c) \land overflow < maxOverflow \land \neg T(c) \implies \Diamond Rw(c))$$

Now the last one is also pretty straightforward, we can use the definition of `receive_worker` we used previously. The full property is as follows:

```
1 assert client_gets_worker {
2   always (
3     all c1 : Client {
4       (c1 in (ClientSendCheckout.cli - Timeout.c)
5       and lt[Poolboy.overflow, MaxOverflow])
6         implies (eventually receive_worker[c1])
7   }
8 }
```

Line 4 of the above listing may be a bit confusing so let us break it down. What we are saying is client `c1` is the client who sent the checkout (take a look at the definition of `ClientSendCheckout` in Appendix A) but does not have any `Timeout` messages.

3.2 ANALYZING THE MODEL

In the last section, we have defined three main properties that are to be verified in Electrum. However, we have not shown how to actually analyze them. In this section we will examine how one can analyze a model in Electrum.

There are two main ways to analyze a model. Electrum can give us instances that satisfy a given formula with the command **`run`**, or present us with counter-examples that contradict some formula with the command **`check`**.

3.2.1 *Validation*

As we explained before, the run command will give us some traces that satisfy a given formula. If no formula is given Electrum will just give us a random valid trace.

```
run {}
```

We can click "Execute" so that Electrum calculates a valid trace, and "Show" to present us a GUI with a graphviz representation of the various signatures an their relations with other signatures spanning multiple time instances. This representation can be themed to best suit the model.

We can force a trace to have a specific number of signatures by adding *scopes*. For instance, suppose say that we want a valid trace where top-level signatures have a maximum cardinality of 3 except for the set `Client` which has exactly 2. In Electrum we could request such a trace as follows:

```
run {} for 3 but exactly 2 Client, 14 Event
```

If no scope is given, it defaults to 3. We can have even more control over the cardinality of our signatures. For example,

```
run {} for 3 but exactly 2 Client, 3..4 Worker, 6 Time, 14 Event
```

Here we are saying that all signatures have a cardinality of 3 except `Client`, which has a cardinality of 2, and `Worker`, which has a cardinality between 3 and 4. **`Time`** is a reserved keyword that represents the size of the path, which is very useful for bounded model checking.

It is also possible to name runs. Suppose we want an instance that, in its initial state, has 2 available workers:

```
run initial_size_two{
  Poolboy.size = 2
}
```

We can even write whole formulas inside a run. Suppose we want a trace where some client will eventually receive a worker. We can reuse the predicate `receive_worker` and specify the run as follows:

```
run eventually_receive {
  some c : Client | eventually receive_worker[c]
}
```

Finally, it is possible in Electrum to force a sequence of steps in a trace using the operator `;`. For instance, if we want a trace where some client first issues a checkout, then receives a worker and finally checks it back in, we can specify that trace as:

```
run specific_trace{
  some ClientSendCheckout ;
  some PbCheckoutReady ;
  some ClientSendCheckin
}
```

The run above only specifies that in the first state some `ClientSendCheckout` event occurs, then in the next state we have `PbCheckoutReady` and finally, in the next third step, we have `ClinetSendCheckin`. There is nothing explicitly stating that the client issuing the checkout is the same as the one checking in. However, since in this model only one event can occur at a time (remember the declaration of `Event`), the client must be the same since no other client has workers to checkin.

3.2.2 *Verification*

In the last section we translated a set of properties from ordinary English into Electrum. To recap, we specified `maximum_workers`, `client_gets_worker`, and `timeout_is_safe`. We now are going to verify if our model satisfies those formulas.

In order to check any formula we use the keyword **`check`**. Like we did with **`run`**, we can define scopes and other constraints. So, if we want to check assertion `maximum_workers`, we write

```
check maximum_workers
```

We can run it by going to "Execute" > "Check maximum_workers", on the right-hand side panel we can read "*Executing "Check maximum_workers"*" and, after done executing, something like *No counter-example found. Assertion may be valid.*. This does not prove that `maximum_workers` is valid in our model, only that Electrum could not find any counter-examples. We can try checking with a different scope, but Electrum still cannot find any counter-examples, which is a pretty good indication that our model, most likely, satisfies `maximum_workers`.

Now consider the property `client_gets_worker` that we defined previously. Although the property is well specified and our model, in theory, does not break it, nevertheless Electrum gives us a counter-example. Let us take a look at it. In Figure 3 we can see a themed representation of the first state as provided by Electrum.

The full trace is as follows:

TIME 0: Client2 sends a checkout

TIME 1: Client1 sends a checkout

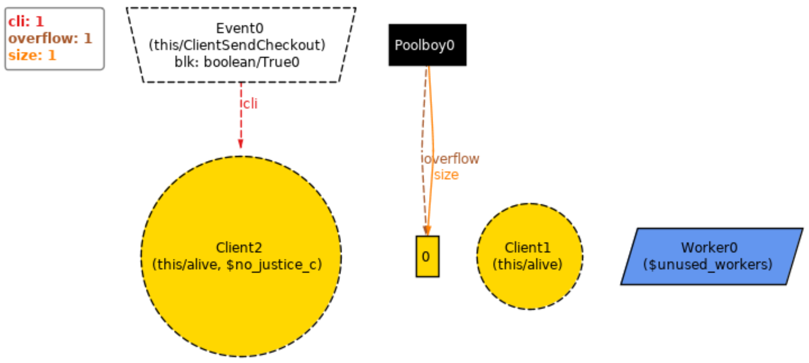


Figure 3: Electrum's representation of the first state

TIME 2: Poolboy reads the last checkout and gives Client1 a worker

TIME 3: Client1 checks in the received worker

TIME 4: Poolboy reads the checkin message from Client1

TIME 5: *loops back to time 1*

Note that Client2 will never get a worker because its message will never be read. Starvation is a classical problem when verifying liveness properties in distributed systems.

In the real word, this is not an issue with Poolboy because the mailbox is implemented with a queue. However, we abstracted that in our model so as to not limit other practical solutions that too are able to guarantee a starvation-free run.

We are not interested in finding a solution for this problem, but we need to force the model to only show us fair and realistic executions. Therefore, we will limit the model behaviour by adding the following fact:

Every message that it is sent will eventually be read.

This is called a *fairness constraint* and it is specified in Electrum as follows:

```
fact ensure_fairness {
  always ( all m : Message {
    sendMessage[m] implies (eventually readMessage[m])
  })
}
```

If we check the property again, we will find that electrum gave us another counter-example. In this new example starvation is still a problem. Instead of the message remaining unread in the mailbox, Poolboy now reads the mailbox and, because it is full, keeps Client2 waiting forever. To solve this, we need to extend our fairness constraint to force every waiting client to get served. The complete Electrum fact now becomes:

```
fact ensure_fairness {
  always ( all m : Message {
    sendMessage[m] implies (eventually readMessage[m])
  })

  always ( all c : Client {
    c in Poolboy.waiting implies (eventually c not in Poolboy.waiting)
  })
}
```

Finally, if we check the liveness property Electrum will not give us any counter-example.

Counter-examples

Finally we need to check assertion `timeout_is_safe`.

This will actually give us a counter-example which may be realistic. In the counter-example, a client issues a checkout and immediately times out, Poolboy first reads the checkout message and sends a worker to the client. Here it broke the property.

This is the exciting part in model checking and why model checking is useful. It is possible that Poolboy might have the same problem so we need to put effort in testing that scenario or looking carefully at the source code. If Poolboy is found to have that problem then it must be fixed; on the other hand, if Poolboy is found to not have this problem, then our model is not accurately representing the system and must be changed. In this case, Poolboy really did have a timeout problem. Thompson resolved this issue by keeping track of the clients that timeout and removing them from the waiting queue.

3.3 BACKEND OVERVIEW

In the past sections we have been talking about how to specify, analyze, and verify a system with Electrum. Now, in this section, we will take a look under the hood to understand how Electrum itself works.

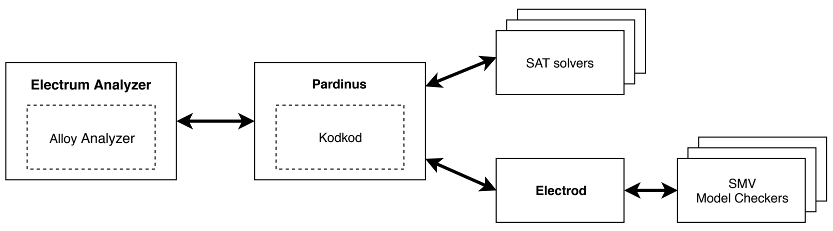


Figure 4: Electrum's architecture

The *Electrum Analyzer* is tool that is responsible for the analysis of the specifications and render instances. It is built on top of Alloy's Analyzer which shares many of the same features.

Alloy's Analyzer uses Kodkod to analyze and provide visual traces, Electrum on the other hand, uses Pardinus, an expansion of Kodkod with temporal relational logic. Pardinus — likewise Kodkod — has a plug-in architecture, which allows us to integrate with other solvers and model checkers. This way, it is possible for the Electrum Analyzer to support both automatic bounded model checking using SAT and unbounded model checking with SMV. Pardinus thus serves as a middleman between the Analyzer and various solvers and model checking tools, thus making it easier for the Analyzer to provide a uniform representation of instances and counter-examples.

When given a linear temporal relational logic specification, Pardinus will first convert it to relational logic.

The complete model checking engine of Pardinus uses a tool called Electrod to convert temporal relational logical formulas of Electrum into plain temporal logic so that they can be model checked by SMV. This is done by deducing the system transitions from the Electrum model and represent them in a `TRANS` section of SMV, as well as other initial conditions and invariants that are possible to derive from the model.

If SMV is successful in finding a trace, then Electrod will pass it to Pardinus which in turn will pass it to Electrum Analyzer to be shown as a graphviz representation of the instance.

4

LTSMIN

LTSmin is a language-independent model checking tool capable of both symbolic and explicit-state analysis. Equipped with multi-core algorithms, it is capable of LTL checking with partial order reduction and symbolic checking for μ-calculus. Its highly versatile nature is thanks to a modular architecture, in which various language front-ends are connected to a model checking core through an interface called *Partitioned Next-State Interface* (**PINS**).

In the following chapter, we will describe in greater detail the PINS architecture and explain the various components that make generic model checking possible. Later we will take a brief look on how it is possible to implement a new language front-end with a bit of C glue code through the DLopen interface. Finally, we will take a look at the innovative way in which partial order reduction is implemented in LTSmin.

4.1 PINS ARCHITECTURE

PINS aims to generalise model checking systems in such a way such that it could be used by very different modelling languages. Therefore, by its very nature, when implementing a front-end to a modelling language it is necessary to implement a few obligatory functions as well as other complementary functions needed for other high performance algorithms.

When we introduced modelling in section 2.1 we explained how a model could be represented by a Kripke structure with:

1. A set of all possible states
2. A set of initial states
3. The transition relation
4. A labelling function

It should then come as no surprise that most of these elements are mandatory functions for LTSmin, namely:

1. InitialState Returns the initial state of the model

2. NextState Returns the successor(s) state(s)
3. StateLabel The labelling function

In LTSmin, a state is represented by a *state vector* with N slots. Actions are grouped into *transition groups*, typically grouping instantiations of the same action with different parameters, which must also be specified; as well as the names and types for each slot in the state vector, transition group and label.

All of these are required for any sort of model checking. But — either for more complex analysis like timed systems or probabilistic models, or to enable high performance algorithms — more information must be provided. In their paper Kant et al. (2015) divide this extra information in a few levels of extensions ($A1$ to $A\infty$) according to their usage. We will only take a look at levels $A1$ and $A2$, as the usage of $A3$ is beyond the scope of this thesis. Table 1 sums up all the extra information as well as the usage for each layer.

The first level we explore is $A1$, the read and write dependency level. Information about these dependencies allow for projections over the state vector, improving performance for both caching, state compression, and symbolic tools; it is also needed to calculate partial order reduction. An action is said to be *read independent* from a slot in the state vector if the action is possible for any value of the slot and the resulting state is the same. Intuitively, an action is said to be *write independent* from a slot in the state vector if the slot is not changed by the action.

The second level down is $A2$, necessary to calculate partial order reduction. It adds more dependency information on transition guards, labels, and actions. *Transition guards* are a subset of all labels; for each transition group, we associate a set of guards — the preconditions for the group to be enabled — which are evaluated conjunctively, that is, all guards must be enabled for the group to be as well. A label is *independent* from a slot in the state vector if changing the slot value does not change the valuation of the labelling function. As we noted in 2.4, the notion of independence between actions is crucial to partial order reduction. This independence is called *accordance* in the terminology used in LTSmin, and a

Level	Used for	Function	Description
$B0$		InitialState	Returns the initial state
$B0$	Basic model checking	NextState	Returns the next state
$B0$		StateLabel	Labelling function
$A1$	Optimizations and	ReadMatrix	Read dependency matrix
$A1$	partial order reduction	WriteMatrix	Write dependency matrix
$A2$		GuardMatrix	Guard/transition group matrix
$A2$	Partial order reduction	StateLabelM	State label dependency matrix
$A2$		DoNotAccord	Actions dependency matrix
$A\infty$	Other usages	GetMatrix	Predefined $X \times Y$ matrix

Table 1: Overview table of the extensions

dependency relation is passed to PINS by a DoNotAccord matrix. We will have a detailed definition of this relation in section 4.3; for now the definition for independence of actions, described before, should suffice.

PINS also allows for other generic matrices to be added and used by GetMatrix. This can be used to improve precision of partial order reduction, although we will ignore it in our development.

This is all the information necessary to calculate partial order reduction. In the next section we will describe a leader election algorithm and we will take a look at how to specify it in PINS with C code via the dlopen interface.

4.2 SPECIFYING IN PINS

Before we try to understand how to translate a language front-end into PINS, we must first figure out how to model a system. Since we also want to test the partial order reduction optimizations of LTSmin, it is best to choose an example with a lot of asynchronous message passing where many actions are independent of one another, in such a way that would maximize the number of interleavings. One such system is the ring leader election which we will describe in the following subsection.

4.2.1 *Showcase Ring leader election example*

In order to test the performance of partial order reduction of LTSmin, we chose to model a system with multiple independent actions that would, without any reduction, generate a myriad of identical traces. For that reason, we have found that a ring election would be the ideal system to model. We chose to specify a ring election where all processes are arranged in a ring and each sends messages to one neighbour (say, to the right) and receives message from the other neighbour (from the left). The full algorithm is described in Algorithm 1.

In our model we will have a fixed number of processes. Each process has an inbox in which it receives messages from other processes; the maximum value they know of; the notion of whether it knows the leader or not; and its current stage – or state –, which tells us if the process is either sending a message or waiting to receive one.

We will now define the initial state and every action that can occur in this system.

Initial state

In the initial state, each inbox is empty, the maximum id known by each process is its own id, no process knows the leader and every process is ready to send.

Algorithm 1 Ring election algorithm for a generic process i

```
1: procedure ELECT
2:     $id \leftarrow getId()$
3:     $max \leftarrow id$
4:     $knowLeader \leftarrow false$
5:     $send_{i+1}(id)$
6:     while $receive_i(mid)$ do
7:         if $mid > max$ then
8:             $max \leftarrow mid$
9:             $send_{i+1}(max)$
10:        else if $mid = max \land \neg knowLeader$ then
11:            $knowLeader \leftarrow true$
12:            $send_{i+1}(max)$
```

Sending

When the process is ready to send, it sends the maximum value it knows of to its neighbour and waits until it has received a new value.

Receiving

The id received by the process can be either smaller than, greater than, or equal to, its maximum id. If a process receives an id that is smaller than its maximum, it just discards the message and awaits a new message. If the received id is greater than the maximum, then the maximum is replaced and propagated. Finally, if the received id is the same as the maximum, that means that we have found our leader a we must propagate the news to our neighbour.

4.2.2 *Modelling ring leader election in C*

Dynamic loading (DL) allows for libraries to be loaded during a program execution. It is, therefore, a very useful technique for implementing extensions to other programs. In UNIX systems, the dlopen API is able to open libraries and prepare them for later use. [1] LTSmin takes advantage of this feature in order to support other language front-ends for PINS; although we can, just as easily, take advantage of this feature to specify any model. For the case of this example, we will use the model we just established, and are now going to specify it in C. First we will take a look at the file structure. Later we will be adding the information that we have previously described in section 4.1.

1 http://tldp.org/HOWTO/Program-Library-HOWTO/dl-libraries.html

File structure

As it is to be expected, we can have as much files as we want; however, to try to keep it simple, we will have just the dlopen implementation in C, as well as a C file where functions such as InitialState and NextState are declared. We will refer to the dlopen implementation C file as simply the *dlopen implementation file* (`dlopen-impl.c`), and to the C file and his header as the *specification file* (`spec.c`) and *header file* (`spec.h`). Depending on what we are modelling, further files can be added as we will later see.

Defining the state, actions and types

As we noted earlier, in PINS the state is represented by a state vector with a fix size of slots. Therefore, in defining the state, we must also specify what each slot represents in our model.

Before we define the state, we must define the existing types of our model. The `lts-type.h` file in the ltsmin library declares a couple of types that can be used. In this example, we need an `int` type, an `action` type for each transition, and a `bool` type for each label and for the `know_leader`. We must also declare the size of our state vector, for that we declare a function in the specification file `state_length`.

```
lts_type_t ltstype=lts_type_create();

// set the length of the state
lts_type_set_state_length(ltstype, state_length());

// add an "int" type for a state slot
int int_type = lts_type_put_type(ltstype, "int", LTStypeSInt32 , NULL);

// add an "action" type for edge labels
int action_type = lts_type_put_type(ltstype, "action", LTStypeEnum, NULL);

// add a "bool" type for state labels
int bool_type = lts_type_put_type (ltstype, "bool", LTStypeBool, NULL);
```

First, because we intend to create a parameterized model, we will declare in our header file a fixed number of processes, in this case 4.

```
#define N 4
```

Although the primary elements in our model are the processes, we must shun from having a slot with all the information regarding a single process, as that would concentrate too much information on a single slot and it would not allow us to make use of valuable high-performance algorithms as partial order reduction. Instead, we will have a slot for each component — i.e., the inbox, the maximum value, the state, and the knowledge of a leader — of each process. We will assume that all process ids are sequential and that they are in the interval $[1, N]$, in that way we can later use the 0 to represent nothing (akin to the null value of programming languages). The inbox will allow processes to send and receive messages from other processes. For now, because LTSmin does not have a "set type" we will have to introduce it later.

We will define function macros in our header file so we can refer to the slot in the state vector later.

```
#define NR_VARS 4

#define INBOX(proc) ((proc - 1) * NR_VARS)
#define MAX(proc) ((proc - 1) * NR_VARS + 1)
#define KNOW_LEADER(proc) ((proc - 1) * NR_VARS + 2)
#define STATE(proc) ((proc - 1) * NR_VARS + 3)
```

In the dlopen implementation file we are required to set a name for each slot and its type.

```
for (int i=1; i <= N; i++) {
    sprintf(name, "inbox_%d", i);
    lts_type_set_state_name(ltstype, INBOX(i), name);
    lts_type_set_state_typeno(ltstype, INBOX(i), int_type);

    (...)
}
```

Similarly, state label types and action types must also be specified. The full specification can be found on appendix B.

The action type must also be populated with the different possible action names:

```
for (int i = 1; i <= N; i++) {
    sprintf(name, "send_%d", i);
    pins_chunk_put(m, action_type, chunk_str(name));

    (...)
}
```

Finally, we validate the type and set it.

```
lts_type_validate(ltstype);
GBsetLTStype(m, ltstype);
```

Initial state, next state and state label

In the initial state every inbox is set to `0`, Since we have not yet declared the "set type" for now we will ignore the initial value of `inbox`, we will add it when introducing the set type. The maximum value is set to the process id. We declare the initial state function in the specification file.

```
int initial[NR_VARS * N];
int* initial_state(void* model) {
    for(int i = 1; i <= N ; i ++) {
        initial[MAX(i)] = i;
        initial[KNOW_LEADER(i)] = 0;
        initial[STATE(i)] = STATE_SEND;
    }

    return initial;
}
```

Finally, we set it as the initial state in the dlopen implementation file.

```
int* initial = initial_state(m);
GBsetInitialState(m, initial);
```

Before we define the next state function, we must first define the various actions that can occur. We divide each action in two functions: the preconditions (called labels) and the postconditions (used in order to calculate the next state). The next state function, declared in the specification file, takes an action group, the state, and a callback function and its arguments. For each possible action we call a function that must calculate the next state.

```
int next_state(void* model, int group, int* src, TransitionCB callback, void* args);
```

In addition to the action label functions, we need to define a goal function that represents the property we want to verify. In this case, our goal is that every process knows the leader, therefore the label goal function is as follows:

```
int label_goal(int* src) {
    int i;
    for(i = 1; i <= N && src[KNOW_LEADER(i)]; i++);
    return i > N;
}
```

Finally, we define the state label function that, for a given label and a state, returns whether the label is valid in that state. It simply uses the previously defined label functions. For simplicity we assume that each action has only a single guard, in that way the event reference (the integer that references the event) is the same for its label. With this in mind, we declare the state label function in our specification file.

```
int state_label(void* model, int label, int* src);
```

And set it in the dlopen implementation file

```
GBsetStateLabelLong(m, (get_label_method_t) state_label);
```

Introducing sets

LTSmin has no default set type, nonetheless it does allow for a generic *chunk type* to be used. We declare the set type in the dlopen implementation file.

```
int set_type = lts_type_put_type(ltstype, "set", LTStypeChunk, NULL);
```

The chunk type can be any serializable type; we used an external set library and defined auxiliary getters and setters for the state vector in our specification file.

```
SSET get_sset(void* model, int* src, int idx);
int set_sset(void* model, int idx, SSET set);
```

Now we can use the above functions whenever we want to take (or put) sets in the state vector. For instance, the initial value for inbox is now:

```
initial[INBOX(i)] = set_sset(model, INBOX(i), sset_init());
```

Dependency matrices

In our specification file, we define the dependency matrices as we described in the previous section 4.1; in the case below, we relate each action with the slots in the state vector read by it.

```
/* Read Matrix
 *                 inbox max know state
 *  SEND             0    1    0    1
 *  RECV_SMALL       1    1    0    1
 *  RECV_EQ          1    1    0    1
 *  RECV_GRT         1    1    0    1
 */
int rm[NR_ACTIONS * N][NR_VARS * N] = { 0 };
void set_read_matrix() {
    for(int p = 1; p <= N; p++) {
        rm[SEND(p)][MAX(p)] = 1;
        rm[SEND(p)][STATE(p)] = 1;

        rm[RECV_SMALL(p)][INBOX(p)] = 1;
        rm[RECV_SMALL(p)][MAX(p)] = 1;
        rm[RECV_SMALL(p)][STATE(p)] = 1;

        rm[RECV_EQ(p)][INBOX(p)] = 1;
        rm[RECV_EQ(p)][MAX(p)] = 1;
        rm[RECV_EQ(p)][STATE(p)] = 1;

        rm[RECV_GRT(p)][INBOX(p)] = 1;
        rm[RECV_GRT(p)][MAX(p)] = 1;
        rm[RECV_GRT(p)][STATE(p)] = 1;
    }
}

int* read_matrix(int row) {
    return rm[row];
}
```

Afterwards, we set them in the dlopen implementation file by copying and setting them. PINS includes some functions to interact with dependency matrices (dm). First, a dependency matrix must be created with `dm_create`. Then its values are set [to true] by the `dm_set` function. Besides the write and read matrices, a combined matrix is also defined which, intuitively, combines the values of both matrices.

```
set_read_matrix();
matrix_t *rm = malloc(sizeof(matrix_t));
dm_create(rm, group_count(), state_length());
for (int i = 0; i < group_count(); i++) {
    int* aux = read_matrix(i);
    for (int j = 0; j < state_length(); j++) {
        if (aux[j]) {
            dm_set(cm, i, j);
            dm_set(rm, i, j);
        }
    }
}
GBsetDMInfoRead(m, rm);
```

This process is similar for other matrices, except for GuardMatrix which is a bit different. Guard matrix relates each transition group with the guards used by it. As we said before, for simplicity we say that each transition group has only one guard which is referenced by the same integer as the action. Therefore, the C code declared in the dlopen implementation is pretty straightforward: For each group there is only one guard with the same reference as the group.

```
guard_t** guards = malloc(group_count() * sizeof(guard_t*));
for(int i = 0; i < group_count(); i++) {
    guards[i] = malloc(sizeof(guard_t) + sizeof(int));
    guards[i]->count = 1;
    guards[i]->guard[0] = i;
}
GBsetGuardsInfo(m, guards);
```

Having completed the model, we are now ready to analyze it. We will see how in the next subsection.

4.2.3 *Analysing with LTSmin*

We now have three C files (plus the additional set library files), we compile them to object files as:

```
gcc -c -I/usr/local/include/ltsmin -I. -std=c99 -fPIC spec.c
gcc -c -I/usr/local/include/ltsmin -I. -std=c99 -fPIC dlopen-impl.c
```

And generate a *shared object*.

```
gcc -shared -o spec.so dlopen-impl.o spec.o
```

We can now pass this shared object to PINS as well as any property we want to check and a file where the counter-example trace will be written. In this case, we want to check that eventually every process will get to know the leader forever. Since we defined `goal` to be the label that is true if every process knows the leader, we can simply call:

```
pins2lts-seq spec.so --ltl="<> goal" --trace=solution.gcf
```

It is also possible to check for invariants. For example, if we want to check that the goal is never achieved, we can call

```
pins2lts-seq espec.so --invariant="! goal" --trace=solution.gcf
```

Because we know that `goal` is eventually possible, the invariant is violated and a counter-example is printed to `solution.gcf`. We can pretty-print `solution.gcf` by running:

```
ltsmin-printtrace solution.gcf
```

It is also possible to use partial order reduction by adding the flag `por`, although we do not yet have the do not accord matrices set up so no reduction must be possible. We will define the do not accord matrix after explaining the partial order reduction algorithm in subsection 4.3.3.

```
pins2lts-seq espec.so --invariant="! goal" --trace=solution.gcf --por
```

LTSmin implements a quite unique partial order reduction algorithm which abstracts away the notion of processes. We will explore this algorithm in the following section.

4.3 GUARD-BASED PARTIAL ORDER REDUCTION

Most partial order reduction algorithms are based on the notion of processes. Some techniques construct program graphs, others rely on internal ordering of the actions of processes. In pursuing a truly language independent model checker, Laarman et al. (2016) developed a partial order reduction algorithm – based on stubborn sets – that is process agnostic. Instead of depending on program counters, their algorithm uses a guard-based approach. Throughout this section we will describe this guard-based partial order reduction.

4.3.1 *A stubborn approach*

The main idea of partial order reduction is to calculate only a *representative subset* of the enabled transitions to explore in a state, thus not generating the whole state graph. This guard-based algorithm is centred on stubborn sets introduced by Valmari. To understand what stubborn sets are, we first need to define a couple of concepts.

As we hinted at in section 2.4, all partial order reduction techniques crucially rely on independence between actions. In guard-based POR, this concept is called *accordance*; We define accordance as:

Definition 1. (Accordance) Two transitions t and t' are said to *accord* if one of the following criteria is true:

1. Their shared variables are disjoint from the write sets;
2. t and t' are never co-enabled;
3. t and t' do not disable each other and their actions commute.

With this, a *do not accord set* ($\mathcal{DNA}$) is defined as the set of transitions pairs that do not accord. We also use $\mathcal{DNA}_t$ to represent the set of transitions that do not accord with t.

In guard-based partial order reduction, the necessary enabling and disabling sets are also a key concept.

Definition 2. (Necessary enabling/disabling set)

1. The set $\mathcal{N}_t$ is said to be a *necessary enabling set* for transition t if and only if for t to be enabled, at least one transition in $\mathcal{N}_t$ must first occur.
2. Conversely, the set $\overline{\mathcal{N}_t}$ is said to be a *necessary disabling set* for transition t if and only if, for t to be disabled at least one transition in $\overline{\mathcal{N}_t}$ must first occur.

A stubborn set is thus defined as:

Definition 3. (Stubborn set) A set $\mathcal{T}_s$ is said to be *stubborn* in state s if, for all $t \in \mathcal{T}_s$:

1. If there is some enabled transition in s, then $\mathcal{T}_s$ must be non-empty;
2. If t is disabled in state s, then there is a transition in $\mathcal{T}_s$ capable of enabling it;
3. If t is enabled in s, then all other transitions that do not accord with t are also in $\mathcal{T}_s$.

Now the partial order algorithm is pretty much any algorithm capable of generating a subset of transitions in a state that matches Definition 3. Algorithm 2 shows us exactly that, by guaranteeing that each new transition added to the set satisfies the conditions above.

In Algorithm 2, $en(s)$ is the set of all enabled transitions in state s, and $find\_nes(t,s)$ is a function that returns a set with all the necessary enabling sets $\mathcal{N}_t$ (there can be many sets for the same transition). We will later see how to calculate the necessary enabling set, for now, let us showcase how the algorithm works.

Algorithm 2 Algorithm to calculate the stubborn set $\mathcal{T}_s$

```
1:  procedure STUBBORN(s)
2:      $\mathcal{T}_{work} \leftarrow \{t\}$ such that $t \in en(s)$
3:      $\mathcal{T}_s \leftarrow \emptyset$
4:      while $\mathcal{T}_{work} \neq \emptyset$ do
5:          choose $t \in \mathcal{T}_{work}$
6:          $\mathcal{T}_{work} \leftarrow \mathcal{T}_{work} \setminus \{t\}$
7:          $\mathcal{T}_s \leftarrow \mathcal{T}_s \cup \{t\}$
8:          if $t \in en(s)$ then
9:              $\mathcal{T}_{work} \leftarrow \mathcal{T}_{work} \cup \mathcal{DNA}_t \setminus \mathcal{T}_s$
10:         else
11:             $\mathcal{T}_{work} \leftarrow \mathcal{T}_{work} \cup \mathcal{N} \setminus \mathcal{T}_s$ where $\mathcal{N} \in find\_nes(t, s)$
12:     return $\mathcal{T}_s$
```

Take a look at Figure 5 which represents a partial run of the guard-based partial order reduction algorithm. For now, t_5 is the only transition in the stubborn set $\mathcal{T}_s$. Now assume that t_5 and t_3 do not accord. In that case, following the algorithm, t_3 will be added to $\mathcal{T}_s$ but, since it is not enabled at state s, some necessary enabling transition must occur, for instance t_2, which is added to $\mathcal{T}_s$. And, in turn t_1 is added to $\mathcal{T}_s$. Suppose $\mathcal{DNA}_{t_1} = \{t_5\}$, in that case, t_5 would be added to the stubborn set, but since it is already there, no change is required. Hence, we end up with the complete set. It shall now be clear how the algorithm is able to secure the conditions of the stubborn set, although we are still missing how to calculate the necessary enabling set; we shall look at exactly that in the next subsection.

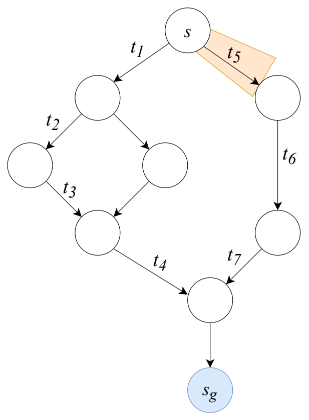


Figure 5: Stubborn set in a mock example.

4.3.2 *Calculating necessary enabling sets*

When introducing the algorithm to calculate the stubborn sets, we assumed the existence of a function $find\_nes(t, s)$ which, given a transition t and a state s, returns the set of all necessary enabling set of t in s. We will now learn how such set can be calculated.

Before we begun reasoning on necessary enabling sets for *transitions*, we must first reflect on necessary enabling sets for *guards*. The definition may seem familiar.

Definition 4. (Necessary enabling set for guards) The set $\mathcal{N}_g$ is said to be a *necessary enabling set for guard g* if and only if, for g to be true, at least one *transition* in $\mathcal{N}_g$ must first occur.

The first question we must ask is "what makes a transition enabled?". The transition is only enabled if all of its guards are satisfied. This means that, for any transition to be enabled, at least a currently disabled guard must become enabled. Therefore, for any transition t and guard g, if g is one of the guards of t then $\mathcal{N}_g$ is also a $\mathcal{N}_t$.

LTSmin statically computes a necessary enabling set for each guard by just considering those transitions that write on variables read by the guard. And $find\_nes(t,s)$ can thus just be some $\mathcal{N}_g$ for some guard g of transition t disabled in state s.

Although using a necessary enabling set is enough to calculate $find\_nes(t,s)$, it often ends up generating large stubborn sets. Take the example presented in Figure 6 where two concurrent processes and their transitions are presented. Suppose both t_1 and t_5 do not accord with t_6. Initially both t_1 and t_6 are enabled and, since they do not accord with each other, both will end up in the stubborn set. Now, because they do not accord, t_6 adds the currently disabled t_5. Working backwards, t_5 adds t_4, et cetera. In the end, we finish with a large stubborn set with t_6, t_1, t_5, and everything in between.

Observe that, to enable a disabled transition one must first disable a transition that cannot be co-enabled with it. For instance, in the above example, t_5 and t_1 cannot be co-enabled, t_1 must be disabled to enable t_5. Therefore, a necessary *disabling* set for t_1 is also a necessary *enabling* set for t_5. In that example, since t_1 is a necessary disabling transition of t_1 (it disables itself), t_1 is a necessary enabling transition for t_5. If we now follow the algorithm, we end up with a stubborn set with just t_1, t_5 and t_6.

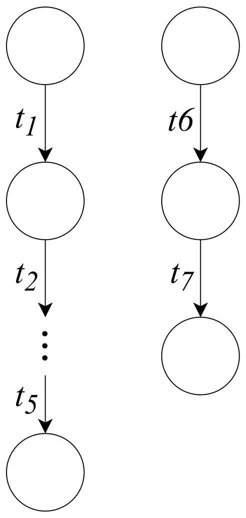


Figure 6: Two transition systems of two concurrent processes.

4.3.3 *Defining the DNA matrix for ring leader election*

When looking at the definition of accordance of transitions, it is possible to understand which transitions accord in our model. We already understand that transitions referring to the same node can never be co-enabled, thus all transition for the same node accord. Regarding transitions of different nodes, they only do not accord if they write on variables of other nodes; the only transition that writes on the slots that are read by other processes is the SEND transition. This way, it very simple to understand that SEND from one process does not accord with the receiving actions of its neighbour. Thus, we have a do not accord such as:

```c
int dnam[NR_ACTIONS * N][NR_ACTIONS * N] = { 0 };
void set_dna_matrix(int row) {
  for(int p = 1; p <= N; p++) {
    dnam[SEND(p)][RECV_SMALL(NEXT(p))] = 1;
    dnam[SEND(p)][RECV_EQ(NEXT(p))] = 1;
    dnam[SEND(p)][RECV_GRT(NEXT(p))] = 1;

    dnam[RECV_SMALL(NEXT(p))][SEND(p)] = 1;
    dnam[RECV_GRT(NEXT(p))][SEND(p)] = 1;
    dnam[RECV_EQ(NEXT(p))][SEND(p)] = 1;
  }
}

int* do_not_accord_matrix(int row) {
  return dnam[row];
}
```

	Time (seconds)		Nr. of States	
nr nodes	w/o POR	w/ POR	w/o POR	w/ POR
4	0	0	152	125
5	0	0,01	425	309
6	0,02	0,03	1732	1147
7	0,07	0,07	3032	1707
8	0,18	0,15	7917	3897
9	0,52	0,37	20527	8777
10	1,76	1,18	53024	19547
11	6,67	2,7	136753	43119
12	20,58	8,12	352627	94341

Table 2: Tests for property eventually everyone knows the leader

4.4 RESULTS

Having extended the rest of the model to allow for partial order reduction, we can now test the improvements of LTSmin. We will look to check the optimizations in time and the reduction of the overall state space for different sized rings. The results we got are shown on Table 2.

This shows promising results but we must keep in mind that this was a manually written model and that a syntactic analysis might not be as precise as our definition of the do not accord.

5

TAINO: AN ELECTRUMESQUE FRONT-END FOR LTSMIN

In order to test the viability of implementing LTSmin's partial order reduction in Electrum, we have decided to implement a new language front-end: Taino named after the lost tribe that inhabited what is today the Caribbean. This new language is based on Electrum syntax; however, due to some restraints of PINS, the semantics will be a bit different.

In this chapter we will present the new language and its differences to standard Electrum already described in Chapter 3. We will also be proposing a new event syntax for Electrum based on the *action layer* suggested by Brunel et al. (2018). Later we will show how we can compile a Taino specification to PINS via the *dlopen* interface. Finally we will present some initial results of this approach.

5.1 THE LANGUAGE

The specification language for Taino is a subset of Electrum extended with a new event syntax. One can declare signatures and relations for signatures. In order to better show how to specify Taino models, we will present a model for a leader election protocol already described in subsection 4.2.1.

5.1.1 *Modelling ring leader election*

In our system we have nodes with a state which describe whether the node is going to receive or send a message. Each node also has the successor node, the maximum value received, and an inbox which is a set of messages. Since each message simply consists of a number, we can represent a message by that number itself; therefore, the inbox can be just be a set of integers.

We define the signature `Node` and other signatures as:

```
enum State { Send, Receive }

sig Int {}

sig Node {
   var max : one Int,
   succ : one Node,
   var inbox : set Int,
   var state : one State
}
```

The above declarations are fairly simple to understand as we already described signatures when we presented the Electrum language. Note that we do not declare `Node` to be variable, since the number of Nodes in the system will not change.

Furthermore, our model also needs to represent the elected nodes. In this case, we make a variable signature:

```
var sig elected {}
```

At present, Taino does not have hierarchy of signatures, which shall not be a problem for this model. In fact, because of the way we translate atoms to C — which we will explore later —, every set is a set of integers. For now, we will simply assume that every element in the `elected` set will be a node.

Declaring signatures in Taino is identical to doing so in Electrum. However, even though Electrum has the notion of time, there is no explicit way to declare events happening in the system. Such omission was also noticed by Brunel et al. (2018), they proposed a new action layer to describe a transition from one state to another. As they note, this action layer is purely syntactic sugar and can easily be converted to regular Electrum. In Taino, we also define a way to represent a transition in the system. However, it does not simply provide an easier and cleaner way of writing a specification; this notion of events is crucial to PINS and therefore to calculate partial order reduction.

An event is parameterizable, meaning some events are related to particular atoms of the system. For instance `send` is an event of a particular `Node`. We declare it as:

```
event send[n : Node] {
  n.state = send

  n.succ.inbox’ = n.succ.inbox + n.max
  n.state’ = receive
}
```

The full declaration of `send` might seem fairly similar to that in Electrum. But whilst in Electrum there is a need to declare the frame conditions — i.e., the variable relations that do not change —, in Taino it is assumed that nothing changes besides those relations explicitly stated in `send`. So, in this case, only `n.succ.inbox` and `n.state` change, everything else in the system remains static. Furthermore, the Taino modelling language is a lot restricted when compared to Electrum; a relation can never be referenced alone, it must always be projected over some atom of its domain.

Taino does not yet support the keyword **some**, therefore the receiving events must be parameterized with the received message, and the pre-condition must check whether such message is actually part of the node's inbox or not. We can see this in action in the definition of `receiveEqual`.

```
event receiveEqual[n : Node, m : Int] {
  n.state = Receive
  m in n.inbox
  n.max = m

  elected' = elected + m
  n.inbox' = n.inbox - m
  n.state' = Send
}
```

Besides the **event** declaration, we also introduce a new declaration that is not available in regular Electrum. This is the **init** declaration where the initial state of the system is specified. This keyword is also introduced to simplify the transition to PINS. Because of some restrictions of LTSmin, it is not possible to define multiple initial states and the definition of **init** must be deterministic. Although in theory this restriction could be overcome by adding a non-deterministic transition from a blank state to all possible initial states.

In Taino, the initial state is declared as follows:

```
init {
  no elected

  Node$1.max = 1
  Node$1.succ = Node$2
  no Node$1.inbox
  Node$1.state = Send

  (...)
} for 4 Node, 4 Int
```

`Node\$1` refers to an atom of the signature Node, it simply refers to the first node. If there are multiple definitions for the same relation Taino will stick to the last defined, except if the relation is constant in which case the first definition is the one that remains. This has to do with the way the init is translated to LTSmin which will be revealed in the next section. Please note that the semantics of the scope definition is slightly different to that of Electrum; in Taino we are stating that there are **exactly** four nodes and four ints in this model.

Finally, we will write predicates to check our goal. In this case, we want to check that eventually forever one (and only one) Node will be elected. We do this by defining a predicate and then checking it. As we already mentioned, most model checkers have a different language for describing the system and for specifying properties over the system.

Although both Alloy and Electrum are exceptions to this rule, LTSmin is not; which makes it impossible to use LTL operators inside predicates, events, facts, or functions. We allow for LTL properties to be declared inside the `check` and, contrary to Electrum, Taino only allows for one `check` to be declared. A check must always reference the predicate to be tested and can also have temporal operators.

```
pred only_one_elected {
  one elected
}

check {
  eventually always only_one_elected
}
```

The full specification can be found at Appendix C, and it shall be quite simple to anyone familiarized with the syntax of Electrum or Alloy. Now, in the following section, we show how this model is translated into LTSmin.

5.2 TRANSLATING TO LTSMIN

Translating a highly abstract modelling language to LTSmin requires us to reason about very minute details and confront the hard differences between these two tools. During this section we will be describing how we can store the Taino notion of the state in the state vector of LTSmin and introduce a mode of indexation. We later use these concepts to initialise the state and to define a next state function. Finally we will show how we can retrieve dependency information purely from a static analysis of the modelling language.

5.2.1 *Defining the state*

The most important element we must reason about is the state; which, due to being arguably the least flexible element of PINS, requires a lot of attention. Recall that for PINS the state is a fixed size vector of values.

We must find a way such that we can transpose the Taino/Electrum notion of state to this fixed sized vector. The relations and signatures which are not variable need not be considered as they can be expressed as C macros, enumerators, or functions. For instance, because static signatures do not change over time, they are not represented in the state, but are rather represented by an enum. Variable sets can be described by equating each set with a slot in the state vector; in the ring leader election, for instance, the `elected` set can be represented by just a slot in the vector. We are still left with the problem of defining the state for variable relations. At first it is tempting to equate each relation to each slot on the state vector. For instance, we would have a slot for `max` that would be a set of relations, something like, following the notation presented when we introduced Electrum,

$$max = \{\langle Node_1, 1\rangle, \langle Node_2, 2\rangle, \ldots, \langle Node_n, n\rangle\}$$

This seems like a reasonable suggestion, but it would mean that every action on `max`, even if taken by different nodes, would change the same slot in the state vector. Take event `receiveGreater`[1] for instance. We understand that events `receiveGreater` of two different nodes are completely independent as they only access the part of the relation belonging to each respective node. However, when the complete relation `max` is in a single slot, this means that any event changing it is conflicting with any other event also accessing it. Thus, if we have a slot of the state vector dedicated to each relation, we could not take advantage of partial order reduction, as pretty much no event would accord with each other. A finer partition must be established to ensure that partial order reduction can still be applied.

Take a look at the definition of `max` again. Even though the relation is marked as variable, the only part of it which really changes is the right-hand part. We can therefore omit the left-hand element of the relation and store the value of `max` for each `Node` in different state slots. We apply this rationale to every relation declared in the model, thus we have a state akin to the one depicted in Figure 7. We make it easier to refer to a specific slot in the state vector, by combining C macros with enumerators. This makes the translation from one language to the other much smoother. In the following example, we are defining all state slots in an enum for a scope of two nodes to make the example smaller.

```c
typedef enum state_names {
  NODE1_MAX,
  NODE1_INBOX,
  NODE1_STATE,
  NODE2_MAX,
  NODE2_INBOX,
  NODE2_STATE,
  ELECTED,
  STATE_LENGTH
} STATE_VAR;

#define size_of_Node 3

#define Node(i) (NODE1_MAX + ((i-1) * size_of_Node))
#define max(i) (Node(i) + 0)
#define inbox(i) (Node(i) + 1)
#define state(i) (Node(i) + 2)
#define elected ELECTED
```

1 see Appendix C.

Node$1.max
Node$1.inbox
Node$1.state
Node$2.max
Node$2.inbox
Node$2.state
elected

Figure 7: Representation of the state vector

The resulting configuration of the state reveals the restriction to the Taino join operator (.) which is used to refer to specific slots in the state vector and does not have the full expressiveness of the set operator of Electrum and Alloy. This, at least in this implementation, severely restricts the expressiveness of an Taino when compared to Electrum.

5.2.2 *Initial state*

Having understood the definition of the state in LTSmin, figuring out how to set it initially is not much more complex. Take a look at the definition of **`init`** in our model (Appendix C). First, the `elected` set is declared to be empty, this simply means that we have to initialise its slot with an empty set. As for simple attributions as is the case of `max`, the value on the analogous slot of the state vector is set to that in the model.

Although the definition of the initial state is pretty straightforward, not everything declared inside the body of **`init`** belongs to the state vector. You may also have noticed that constants are also defined as well as the scopes for each signature.

As for scopes, we can just use enumerators to refer to specific atoms, in such a way that `Node$1` would translate to `NODE1`, hence a scope for four nodes translates to:

```
typedef enum Node{
  NODE1 = 1,
  NODE2,
  NODE3,
  NODE4
} NODE_T;
```

In the case of constant relations, as in `succ`, a function is defined that receives the left-hand side of the "join operator" (.) and returns the corresponding value. Taino does not allow for constant relations of different multiplicities. The `succ` constant relation can thus end up looking something like:

```
int succ(int i){
  switch(i) {
    case NODE1:
      return NODE2;
    case NODE2:
      return NODE3;
    case NODE3:
      return NODE4;
    case NODE4:
      return NODE1;
  }
  return 0;
}
```

In this way, we can later refer to the slot for the inbox of the successor of `Node$1` (`Node$1.succ.inbox`) as `inbox(succ(NODE1))`. The restrictions of this "join operator" may now be clearer, as it is purely used to index the state vector and not as a join of two sets.

5.2.3 *Defining Next state*

The indexation by enumerators and C macros described in the previous subsections are now very useful to translate events into actual LTSmin. But we also need to enumerate all transitions possible, for that we will through each event and parameterize it to create an enum.

```
typedef enum trans_labels {
  RECEIVEEQUAL_NODE1_INT1,
  RECEIVEEQUAL_NODE1_INT2,
  (...)
  RECEIVEEQUAL_NODE4_INT1,
  RECEIVEEQUAL_NODE4_INT2,
  (...)
  SEND_NODE4,
  LABEL_GOAL
} TRANSITIONS;
```

You can see that the number of transitions grows exponential to the number of parameters for each event.

In Taino the pre-conditions and post-conditions of each event are described together, LTSmin however isolates transitions from its guards. Therefore, each **event** translates to two functions: one for the label and another for the transition itself. In Taino, we assume that each transition only has one guard which is the conjunction of all pre-conditions.

For a specific example, let us look at event `receiveGreater`. It generates the following label function:

```
int label_receive_greater(void* model, int* src, int n, int m){
    // n.state = Receive
    int p1 = src[state(n)] == RECEIVE_T;

    // m in n.inbox
    SET n_inbox_set = get_set(model, src, inbox(n));
    int p2 = set_exists(n_inbox_set, m);
    set_free(n_inbox_set);

    // gt[m, n.max]
    int p3 = gt(m, src[max(n)]);

    return p1 && p2 && p3;
}
```

The comments are added here to better show what relates to the Taino specification. Note that functions like `get_set` allocate the set in memory and `set_free` ensures that the memory is safely freed as soon as it is no longer needed; this is the reason for declaring a temporary `n_inbox_set`.

The label function uses the snake case version of the event name given in the model with a `label_` prefix, this is the standard for all label functions of events. Apart from the reference to the model and to the state vector (`src`), the label function also receives the arguments of the event, in this case the node `n` and message `m` which are represented by integers.

The function that calculates the next state for this event is not much more complex, in fact it takes advantage of the label function to confirm that the event is going to happen. The mindset behind the translation of the post-conditions is that it simply changes the state vector with new values. This is why the syntax for events in Taino is so restrictive, it expects to assign new values to slots in the state vector. Besides the arguments that are received by the label function, this function also receives the destination state vector – i.e., the state vector of the following state – and a copy vector (`cpy`) which tells PINS which slots in the state vector have not changed and can thus be copied. This copy vector is used by PINS to perform some optimizations.

```
int specm_receive_greater(void* model, int* src, int* dst, int* cpy, int n, int m){
    int succs = 0;
    SET n_inbox_dst;
    SET m_set;
    SET n_inbox_set;

    if (label_receive_greater(model, src, n, m)) {
        // n.max' = m
        dst[max(n)] = m;

        // n.inbox' = n.inbox - m
        n_inbox_set = get_set(model, src, inbox(n));
        m_set = set_init_one(m);
        n_inbox_dst = set_difference(n_inbox_set, m_set);
        set_free(n_inbox_set);
        set_free(m_set);
        dst[inbox(n)] = set_set(model, inbox(n), n_inbox_dst);
        set_free(n_inbox_dst);

        // n.state' = Send
        dst[state(n)] = SEND_T;

        cpy[max(n)] = 0;
        cpy[inbox(n)] = 0;
        cpy[state(n)] = 0;

        succs++;
    }
    return succs;
}
```

The comments were also added to highlight the translation from the Taino model to C. Like the label function, this function also ensures that all allocated memory is freed as soon as it is no longer needed.

Taino does not yet support non-determinism in actions; but in order to support it, instead of these three vectors, the function should receive three vectors of vectors (for each resulting state) and return the number of successor states (`succ`) – which is incremented as each new successor state is constructed. Here, because we do not have non-determinism, the `succ` variable is incremented just once.

You can see the powerful application of the indexation scheme we have been describing, it not only makes the translation easier, it also ensures the resulting C code is human readable.

With these functions, the NextState function of LTSmin, simply checks for which action is happening and calls the respective event function with its arguments.

5.2.4 *Calculating dependency matrices*

As we noted when we introduced LTSmin back in chapter 4, the dependencies are crucial to calculate partial order reduction, so we must ensure that we can get all information needed by a syntactic analysis.

It is very clear how to get the information needed to calculate read and write dependencies with events. Each event is composed of various expressions. Thus, to calculate read dependencies, one must only look for the elements of the state vector which are in the expression, ignoring the ones with an apostrophe `'`. Whilst for write dependencies we simply look to the expressions with the apostrophe.

With is information, it is possible to generate a function that sets the read and write dependency matrices, like the excerpt below:

```c
int wm[LABEL_GOAL][STATE_LENGTH] = { 0 };
void set_write_matrix() {
    wm[RECEIVEEQUAL_NODE1_INT1][ELECTED] = 1;
    wm[RECEIVEEQUAL_NODE1_INT1][NODE1_INBOX] = 1;
    wm[RECEIVEEQUAL_NODE1_INT1][NODE1_STATE] = 1;
    wm[RECEIVEEQUAL_NODE1_INT2][ELECTED] = 1;
    wm[RECEIVEEQUAL_NODE1_INT2][NODE1_INBOX] = 1;
    wm[RECEIVEEQUAL_NODE1_INT2][NODE1_STATE] = 1;
    ...
}
```

The calculation for accordance is significantly more complex. Recall the definition of accordance (Definition 1), where for two transitions t and t' to accord one of the following criteria must be met:

1. Their shared variables are disjoint from the write sets;
2. t and t' are never co-enabled;
3. t and t' do not disable each other and their actions commute.

At first, we tried calculating if two events **do not** accord by just looking at their shared variables (criteria 1). However, this definition was not accurate enough and, due to the recursive nature of the stubborn set algorithm (Algorithm 2), would wrongly identify too many according events as not according. This lead to **no** partial order reduction being possible for a simple example like this ring leader election.

We later were able to introduce a very simple analysis to check if two events can be co-enabled. We do this by solely looking at pre-conditions that check for two different values in the slot vector. For instance, we can say that, `send` and `receiveEqual` are never co-enabled for the same node as the pre-condition for `send` is `n.state = send`, whilst in `receiveEqual` we have `n.state = receive`. This, admittedly naïve, analysis was sufficient to define a more precise accordance between two events for this model.

As for the third criteria, we assumed that if two events did not share the same variables, then they could never disable each other, although a method to assert if two events commute is also essential, and requires deeper investigation.

5.3 THE TAINO PROJECT

Taino is written in Haskell and it is responsible to translate the Taino modelling language to LTSmin. This is achieved in three steps: first, the original Taino model is parsed into an abstract syntax tree (AST); this AST is then converted into a more flexible data structure that can be queried to write the three different files (`spec.c`, `spec.h`, and `dlopen.c`). This schema is represented in Figure 8.

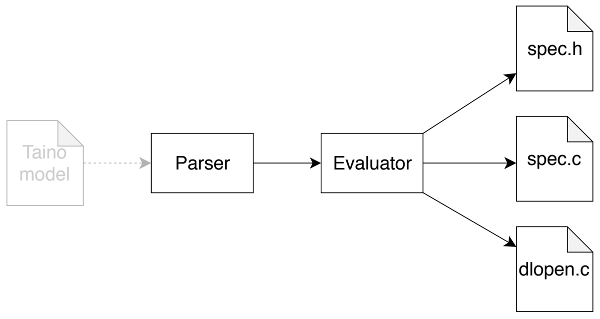


Figure 8: Representation of the Taino architecture

	Time (seconds)		Nr. of states	
nr nodes	w/o POR	w/ POR	w/o POR	w/POR
4	0.01	0.01	160	133
5	0.03	0.05	441	325
6	0.09	0.14	1215	836
7	0.23	0.42	3132	1846
8	0.64	1.02	8045	4025
9	2.31	2.78	20783	9033
10	8.18	11.23	53536	20056
11	29.59	35.46	137777	44143
12	105.35	97.45	354675	96389

Table 3: Tests for property always at most one elected

The parser was implemented using megaparsec[2], which takes de taino model and builds an abstract syntaxt tree. Later, the evaluator splits the abstract syntax tree in the various elements of the model: signatures, enums, events, init, predicates, et cetera. This in turn makes it easier to generate the three files needed. Lastly, we have three models each responsible to generate each file (`spec.c`, `spec.h`, and `dlopen.c`).

5.4 RESULTS

We tested our implementation by checking different properties and different models of the ring leader election. We checked to see how much state reduction we could get out of LTSmin guard-based algorithm, and to see how that reduction would translate into a hopefully shorter time to check the properties.

We have run tests on three properties:

1. Always at most one elected;
2. Eventually always one elected;
3. As soon as everyone agrees on the leader, all mailboxes should be empty.

2 https://hackage.haskell.org/package/megaparsec

	Time (seconds)		Nr. of states	
nr nodes	w/o POR	w/ POR	w/o POR	w/POR
4	0.01	0.02	263	209
5	0.03	0.05	737	505
6	0.11	0.16	2005	1181
7	0.37	0.49	5359	2709
8	1.25	1.49	14169	6129
9	4.69	5.86	37213	13713
10	23.03	17.07	97349	30389
11	85.76	77.6	254049	66781
12	275.11	225.04	662245	145673

Table 4: Tests for property eventually always one elected

The first two properties should be true, LTSmin will not be able to find a counter-example and will have to search the entire state space. The third property will not be true in our model, and LTSmin should find a counter-example relatively easily; we do not expect partial order reduction to be much of an improvement.

Table 3 shows the results for the first property. Even though partial order reduction was able to drastically reduce the number of states, the necessary overhead of calculations for such reduction did not compensate. On Table 3 only with twelve nodes did the reduced state space begun to compensate for the overhead of calculations.

For the second property, which results can be found in Table 4. Again, only with ten nodes does partial order reduction begin to have a positive impact.

Finally, because LTSmin is able to immediately find a counter-example for property 3, the partial order reduction overhead is simply counter-productive. Table 5 shows how much impact the partial order reduction overhead has.

	Time (seconds)	
nr nodes	w/o POR	w/ POR
4	0	0.01
5	0	0.02
6	0.01	0.04
7	0.01	0.06
8	0.01	0.11
9	0.02	0.18
10	0.02	0.27
11	0.03	0.4
12	0.05	0.58

Table 5: Tests for the emptiness of the mailboxes when everyone agrees on the leader

These results show that great reductions in the state-space do not translate directly into a better performance. This is specially true in smaller examples with a small scope where the effects of state-explosion are not very perceptible.

6 CONCLUSION

With the work done and the results gathered, it is now time to look back on the progress we have made and reflect on its significance.

Throughout this chapter we will take some conclusions on the outcome of the whole investigation and reason about the results we have got. We must also look forward and envision what comes next, what work is there still to do and what can grow out of all of this research.

6.1 CONCLUSIONS

Throughout the course of this investigation, we have learned about the various partial order reduction algorithms and their accompanying model checkers. We have studied both dynamic and static partial order reduction algorithms and found their limitations, and we have learned about a novel guard-based partial order reduction which does not need the notion of process and their program counters.

We have successfully implemented an Electrum-like language front-end for LTSmin, and we have described syntactic analysis to calculate dependencies on a language of this type. We have also shown how a relational model can be expressed in a single state vector, as well as the changes to the syntax of Electrum that need to be made in order to consider having partial order reduction, namely events need to be specified explicitly.

However, we have observed that partial order reduction is only effective for state spaces with a large amount of values, and that only with a very detailed analysis can the reduction be possible. Further, we now understand that the overhead of calculations needed for partial order reduction is much grater that what we have anticipated.

6.2 PROSPECT FOR FUTURE WORK

Although we have concluded that partial order reduction does not bring a lot of gains for a small number of variables in the model, this work still opens a lot of doors for the future, as we are not taking full advantage of the algorithm and there are very large models for Electrum where partial order reduction would be helpful. And the gains from the manual model presented back on section 4.4, shows that it is possible to mitigate the overhead produced by the state-space reduction.

There are two main approaches to achieve better usage of partial order reduction and to reduce its necessary overhead of calculations. The first is to develop a more advanced technique to calculate the do-not-accord relation, and the second is to reduce the size of the necessary enabling sets.

As for a more advanced technique to calculate the do-not-accord relation, we suggest that a finner syntactic analysis should be studied. Such analysis must be able to look at pre-conditions with different cardinalities of the same set and opposing comparisons, to determine if two events may be co-enabled; as well as distinguish commutative actions, for example adding a value to a set.

As to reduce the size of necessary enabling sets, first we should aim to reduce the number of overall transitions. In our case study, the ring leader election, a `some` keyword would be helpful since it would remove an event argument and drastically reduce the number of transitions in the system. However that would bring non-determinism to model, the effects of which still need to be studied. Secondly, we propose to attribute multiple guards for each event instead of a single guard.

BIBLIOGRAPHY

R. Alur, R. K. Brayton, T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Partial-order reduction in symbolic state space exploration. In Orna Grumberg, editor, *Computer Aided Verification*, pages 340–351, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69195-2.

Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 229–248, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89963-3.

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2007. ISBN 0-201-85469-4.

Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. ISBN 978-3-540-49059-3.

Julien Brunel, David Chemouil, Alcino Cunha, Thomas Hujsa, Nuno Macedo, and Jeanne Tawa. Proposition of an action layer for electrum. In Michael Butler, Alexander Raschke, Thai Son Hoang, and Klaus Reichl, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 397–402, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91271-4.

J. Richard Büchi. *On a Decision Method in Restricted Second Order Arithmetic*, pages 425–435. Springer New York, New York, NY, 1990. ISBN 978-1-4613-8928-6. doi: 10.1007/978-1-4613-8928-6_23. URL https://doi.org/10.1007/978-1-4613-8928-6_23.

E. Clarke, O. Grumberg, and K. Hamaguchi. Another look at ltl model checking. In David L. Dill, editor, *Computer Aided Verification*, pages 415–427, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg. ISBN 978-3-540-48469-1.

E Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 12 1982. doi: 10.1016/0167-6423(83)90017-5.

Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, January 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040315. URL http://doi.acm.org/10.1145/1047659.1040315.

Jaco Geldenhuys, Henri Hansen, and Antti Valmari. Exploring the scope for partial order reduction. In Zhiming Liu and Anders P. Ravn, editors, *Automated Technology for Verification and Analysis*, pages 39–53, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04761-9.

Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521. URL https://doi.org/10.1109/32.588521.

Daniel Jackson. *Software Abstractions*. The MIT Press, 2006. ISBN 0-262-10114-9.

Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. Ltsmin: High-performance language-independent model checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 692–707, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46681-0.

Saul Kripke. Semantical considerations on modal logic. 1963.

Alfons Laarman, Elwin Pater, Jaco van de Pol, and Henri Hansen. Guard-based partial-order reduction. *International Journal on Software Tools for Technology Transfer*, 18(4):427–448, Aug 2016. ISSN 1433-2787. doi: 10.1007/s10009-014-0363-9.

Leslie Lamport. *Specifying Systems: The TLA+ Language and T ools for Hardw are and Soft w are Engineer*. Addison-Wesley, 2003. ISBN 0-321-14306-X.

Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight specification and analysis of dynamic systems with rich configurations. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 373–383, 2016.

Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993. ISBN 0792393805.

Chris Newcombe. Why amazon chose tla+. In Yamine Ait Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. ISBN 978-3-662-43652-3.

Amir Pnueli. The temporal logic of programs. pages 46–57, 09 1977. doi: 10.1109/SFCS.1977.32.

R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2): 146–160, 1972. doi: 10.1137/0201010.

Andrew Thompson. Quickchecking poolboy for fun and profit, 2012. URL https://vagabond.github.io/programming/2012/01/21/quickchecking-poolboy-for-fun-and-profit.

Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990*, pages 491–515, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg. ISBN 978-3-540-46369-6.

Moshe Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). pages 332–344, 01 1986.

A

POOLBOY SPECIFICATION

```
open util/integer
open util/boolean

one var abstract sig Event {}
abstract sig Actor {}
abstract sig Message {}

one sig MaxOverflow in  Int {}

var sig alive in Actor {}
var sig blocked in alive {}

sig Worker extends Actor {}

sig Client extends Actor {
  var workers : set Worker
}

one sig Poolboy {
  var free : set Actor,   -- Every free actor poolboy has
  var overflow : one Int,   -- The number of overflow workers poolboy gave
  var waiting : set Client, -- Clients waiting for workers
  var mailbox : set Message,  -- Mailbox common to every actor ****
  size : one Int        -- The initial size of free
}

fun unused_workers : set Worker {
  Worker - ( Client.workers + Poolboy.free + alive ) - ( Poolboy.mailbox & Exit ).a
}

------------------   MESSAGES  -----------------------

sig Checkout extends Message {
  c : Client, -- Client issuing the checkout
  b : Bool  -- Blocking or non blocking
```

```
}

sig Checkin extends Message {
  c : Client, -- Client issuing the checkin
  a : Actor -- Checkin actor
}

sig Exit extends Message {
  a : Actor   -- The actor exiting
}

sig Timeout extends Message {
  c : Client
 }

pred sendMessage[m : Message] {
  m not in Poolboy.mailbox
  mailbox’ = mailbox + Poolboy->m
}

pred readMessage[m : Message] {
  m in Poolboy.mailbox
  mailbox’ = mailbox - Poolboy->m
}

--------------------   EVENTS   ------------------------------------------------

var sig ClientSendCheckout extends Event {
  var cli : Client,
  var blk : Bool
} {
  cli in (Client & alive) - blocked

  some msg : Checkout {
    msg.c = cli
    msg.b = blk
    sendMessage[msg]
  }

  blocked’ = blocked + cli
}

var sig ClientSendCheckin extends Event {
  var cli : Client,
  var act : Actor
} {
  cli in (Client & alive) - blocked
  act in cli.workers

  some msg : Checkin {
    msg.c = cli
    msg.a = act
    sendMessage[msg]
  }
  workers' = workers - cli->act
}

var sig ClientSendTimeout extends Event {
  var cli : Client
} {
  cli in blocked

  some msg : Timeout {
    msg.c = cli
    sendMessage[msg]
  }

  blocked' = blocked - cli
}

var sig ActorExit extends Event {
  var act : Actor
} {
  act in Worker -- non byzantine
  act in alive

  some msg : Exit {
    msg.a = act
    sendMessage[msg]
  }

  alive' = alive - act
  blocked' = blocked - act
  workers' = workers - Client->act
}

var sig PbCheckoutReady extends Event {} {
  some free

  some msg : Checkout {
    readMessage[msg]

    some w : Poolboy.free {
      w in alive
      free' = free - Poolboy->w
      workers' = workers + msg.c->w
    }

    blocked' = blocked - msg.c
  }
}

var sig PbCheckoutOverflow extends Event {} {
  no free
  lt[Poolboy.overflow, MaxOverflow]

  some msg : Checkout {
    readMessage[msg]

    some w : unused_workers {
      alive' = alive + w
      workers' = workers + msg.c->w
    }

    blocked' = blocked - msg.c
  }
  Poolboy.overflow' = add[Poolboy.overflow, 1]
}

var sig PbCheckoutFull extends Event {} {
  no free
  (lte[MaxOverflow, 0] or gte[Poolboy.overflow, MaxOverflow])

  some msg : Checkout {
    readMessage[msg]
    isTrue[msg.b] implies Poolboy.waiting' = Poolboy.waiting + msg.c
      else (waiting' = waiting and blocked' = blocked - msg.c)
  }
}

var sig PbTimeout extends Event {} {
  some msg : Timeout {
    readMessage[msg]
    Poolboy.waiting' = Poolboy.waiting - msg.c
  }
}

var sig PbCheckinWaiting extends Event {} {
  some msg : Checkin, cli : Poolboy.waiting {
    readMessage[msg]

    Poolboy.waiting’ = Poolboy.waiting - cli
    workers’ = workers + cli->msg.a
    blocked’ = blocked - cli
  }
}

var sig PbCheckinOverflow extends Event {} {
  no waiting
  gt[Poolboy.overflow, 0]

  some msg : Checkin {
    readMessage[msg]
    alive’ = alive - msg.a
  }
  Poolboy.overflow’ = sub[Poolboy.overflow, 1]
}

var sig PbCheckinReady extends Event {} {
  no waiting
  Poolboy.overflow = 0

  some msg : Checkin {
    readMessage[msg]
    Poolboy.free’ = Poolboy.free + msg.a
  }
}

var sig PbExitWaiting extends Event {} {
  some msg : Exit, cli : Poolboy.waiting {
    readMessage[msg]

    some w : unused_workers {
      alive’ = alive + w
      workers’ = workers + cli->w
    }

    Poolboy.waiting’ = Poolboy.waiting - cli
    blocked’ = blocked - cli
  }
}

var sig PbExitOverflow extends Event {} {
  no waiting
  gt[Poolboy.overflow, 0]

  some msg : Exit {
    readMessage[msg]
    Poolboy.free’ = Poolboy.free - msg.a
  }

  Poolboy.overflow’ = sub[Poolboy.overflow, 1]
}

var sig PbExitReady extends Event {} {
  no waiting
  Poolboy.overflow = 0

  some msg : Exit, w : unused_workers {
    readMessage[msg]
    Poolboy.free’ = Poolboy.free - msg.a + w
    alive’ = alive + w
  }
}

----------------   UPDATE FACTS   ---------------------------------------------

fact updated {
  always ( blocked’ != blocked
    implies some ClientSendCheckout + ClientSendTimeout + ActorExit +
      PbCheckoutReady + PbCheckoutOverflow + PbCheckinWaiting + PbExitWaiting)

  always ( workers’ != workers
    implies some ClientSendCheckin + ActorExit + PbCheckoutReady +
      PbCheckoutOverflow + PbCheckinWaiting + PbExitWaiting )

  always ( alive’ != alive
    implies some ActorExit + PbCheckoutOverflow + PbCheckinOverflow +
      PbExitWaiting + PbExitReady)

  always ( free’ != free
    implies some PbCheckoutReady + PbCheckinReady + PbExitOverflow + PbExitReady)

  always ( overflow’ != overflow
    implies some PbCheckoutOverflow + PbCheckinOverflow + PbExitOverflow)

  always ( waiting’ != waiting
    implies some PbCheckoutFull + PbTimeout + PbCheckinWaiting + PbExitWaiting)

  always ( mailbox’ != mailbox
    implies some ClientSendCheckout + ClientSendCheckin + ClientSendTimeout + ActorExit +
      PbCheckoutReady + PbCheckoutOverflow + PbCheckoutFull + PbTimeout  + PbCheckinWaiting +
      PbCheckinOverflow + PbCheckinReady + PbExitWaiting + PbExitOverflow + PbExitReady)
}

```

```
---------------   FACTS    ---------------------------------------------------

fact init {
  Poolboy.overflow = 0
  gte[MaxOverflow, 0]
  Poolboy.free = alive & Worker
  Poolboy.size = #Poolboy.free
  no waiting
  no workers
  no blocked
  no mailbox
}

--------------------------------------------------------------------------------

fact ensure_justice {
  always ( all m : Message {
    sendMessage[m] implies (eventually readMessage[m])
  })

  always ( all c : Client {
    c in Poolboy.waiting implies (eventually c not in Poolboy.waiting)
  })
  --eventually always no waiting
}

pred workers_available {
  some Poolboy.free or
  lt[Poolboy.overflow, MaxOverflow]
}

pred receive_worker[c : Client] {
  #c.workers' = add[#c.workers, 1]
}

assert maximum_workers {
  always lte[#total_workers, add[Poolboy.size, MaxOverflow]]
}
check maximum_workers for 6

assert client_gets_worker {
  all c1 : Client {
    always (
      (c1 in (ClientSendCheckout.cli - Timeout.c) and lt[Poolboy.overflow, MaxOverflow])
        implies (eventually receive_worker[c1]))
  }
} check client_gets_worker
```

```

assert timeout_is_safe {
  always (
    all c1 : Client {
      receive_worker[c1] implies c1 in blocked
    }
  )
} check timeout_is_safe

fun total_workers : set Worker {
  Poolboy.free + Client.workers
}


run eventually_receive {
  some c : Client | eventually receive_worker[c]
} for 4 but 14 Event

run specific_trace{
  some ClientSendCheckout ;
  some PbCheckoutReady ;
  some ClientSendCheckin
} for 4 but 3 Event

run {
  eventually some PbExitWaiting
} for 4 but 14 Event
```

B LTSMIN MODEL

B.1 specification file header (spec.h)

```c
#ifndef RING_H
#define RING_H

#include "sSet.h"
#include <ltsmin/pins.h>

#define N 6
#define NR_VARS 4
#define NR_ACTIONS 4

#define INBOX(proc) ((proc - 1) * NR_VARS)
#define MAX(proc) ((proc - 1) * NR_VARS + 1)
#define KNOW_LEADER(proc) ((proc - 1) * NR_VARS + 2)
#define STATE(proc) ((proc - 1) * NR_VARS + 3)

#define NEXT(proc) ((proc % N) + 1)
#define PREV(proc) (((proc + (N-2)) % N) + 1)

// transition labels
#define SEND(i) ((i-1) * NR_ACTIONS)
#define RECV_SMALL(i) ((i-1) * NR_ACTIONS + 1)
#define RECV_EQ(i) ((i-1) * NR_ACTIONS + 2)
#define RECV_GRT(i) ((i-1) * NR_ACTIONS + 3)

// label values
#define LABEL_GOAL (NR_ACTIONS * N)

/**
 * @brief calls callback for every successor state of src in transition group "group".
 */
int next_state(void* model, int group, int *src, TransitionCB callback, void *arg);

/**
```

```c
 * @brief returns the initial state.
 */
int* initial_state(void* model);

/**
 * @brief returns the read dependency matrix.
 */
void set_read_matrix();
int* read_matrix(int row);

/**
 * @brief returns the write dependency matrix.
 */
void set_write_matrix();
int* write_matrix(int row);

/**
 * @brief returns the state label dependency matrix.
 */
void set_label_matrix();
int* label_matrix(int row);

/**
 * @brief returns the guard dependency matrix.
 */
void set_dna_matrix();
int* guard_matrix(int row);

/**
 * @brief returns the do not accord matrix.
 */
int* do_not_accord_matrix(int row);

/**
 * @brief returns whether the state src satisfies state label "label".
 */
int state_label(void* model, int label, int* src);

/**
 * @brief returns the number of transition groups.
 */
int group_count();

/**
 * @brief returns the length of the state.
 */
int state_length();
```

```

/**
 * @brief returns the number of state labels.
 */
int label_count();

/**
 * @brief returns the number of guards.
 */
int guard_count();
int* label_indices();

void get_guard_all(void* model, int* src, int g, int* labels);
SSET initial_inbox(int proc);
#endif
```

B.2 SPECIFICATION FILE (SPEC.C)

```
#include "spec.h"

#include <ltsmin/pins-util.h>

/*****************************
 *    RING ELECTION
 * This is a model for a ring
 * election for N nodes
 *****************************/

// state values
static const int STATE_SEND = 0;
static const int STATE_RECEIVE = 1;

int ring_send(void* m, int from, int to, int* src, int* dst, int* cpy);
int receive_equal(void* m, int proc, int* src, int* dst, int* cpy);
int receive_greater(void* model, int proc, int* src, int* dst, int* cpy);
int receive_smaller(void* model, int proc, int* src, int* dst, int* cpy);

int label_goal(int* src);
int label_send(int p, int* src);
int label_receive_smaller(void* m, int p, int* src);
int label_receive_equal(void* m, int p, int* src);
int label_receive_greater(void* m, int p, int* src);

SSET get_sset(void* model, int* src, int idx) {
  lts_type_t lts_type = GBgetLTStype(model);
  int set_type = lts_type_get_state_typeno(lts_type, idx);
  chunk chunk_inbox_from = pins_chunk_get (model, set_type, src[idx]);

  SSET ret = sset_deserialize(chunk_inbox_from.data, chunk_inbox_from.len);
  int test[1024], size = sset_to_list(ret, test, 1024);

  return ret;
}

int set_sset(void* model, int idx, SSET set) {
  char data[1024];
  lts_type_t lts_type = GBgetLTStype(model);
  int set_type = lts_type_get_state_typeno(lts_type, idx);
  int length = sset_serialize(set, data, 1024);

  int id = pins_chunk_put(model, set_type, chunk_ld(length, data));

  int test[1024], size = sset_to_list(set, test, 1024);

  return id;
}

int group_count() {
    return NR_ACTIONS * N;
}

int state_length() {
    return NR_VARS * N;
}

int label_count() {
    return NR_ACTIONS * N + 1;
}

int guard_count() {
    return NR_ACTIONS * N;
}

int next_state(void* model, int group, int* src, TransitionCB callback, void* args) {
  int i, succs = 0;
  int dst[state_length()], cpy[state_length()];
  int action[1];

  memcpy(dst, src, state_length() * sizeof(int));
  memset(cpy, 1, state_length() * sizeof(int));

  transition_info_t transition_info = { action, group };
```

```c

  action[0] = group;

  for(i = 1 ; i <= N && !succs; i++) {
    if (group == SEND(i)) {
      succs = ring_send(model, i, NEXT(i), src, dst, cpy);
    } else if (group == RECV_SMALL(i)) {
      succs = receive_smaller(model, i, src, dst, cpy);
    } else if (group == RECV_EQ(i)) {
      succs = receive_equal(model, i, src, dst, cpy);
    } else if (group == RECV_GRT(i)) {
      succs = receive_greater(model, i, src, dst, cpy);
    }
  }

  if (succs) {
    callback(args, &transition_info, dst, cpy);
  }

  return succs;
}

int initial[NR_VARS * N];
int* initial_state(void* model) {
  for(int i = 1; i <= N ; i ++) {
    //initial[INBOX(i)] = 0;  // inbox proc_i
    initial[INBOX(i)] = set_sset(model, INBOX(i), sset_init());
    initial[MAX(i)] = i;  // max proc_i
    initial[KNOW_LEADER(i)] = 0;  // know_leader proc_i
    initial[STATE(i)] = STATE_SEND; // state proc_i
  }

  return initial;
}

SSET initial_inbox(int proc) {
  return sset_init();
}

/* Read Matrix
 *          inbox max know state
 *  SEND        0    1    0    1
 *  RECV_SMALL    1    1    0    1
 *  RECV_EQ     1    1    0    1
 *  RECV_GREATER  1    1    0    1
 */
int rm[NR_ACTIONS * N][NR_VARS * N] = { 0 };
```

```
void set_read_matrix() {
  for(int p = 1; p < N; p++) {
    rm[SEND(p)][MAX(p)] = 1;
    rm[SEND(p)][STATE(p)] = 1;

    rm[RECV_SMALL(p)][INBOX(p)] = 1;
    rm[RECV_SMALL(p)][MAX(p)] = 1;
    rm[RECV_SMALL(p)][STATE(p)] = 1;

    rm[RECV_EQ(p)][INBOX(p)] = 1;
    rm[RECV_EQ(p)][MAX(p)] = 1;
    rm[RECV_EQ(p)][STATE(p)] = 1;

    rm[RECV_GRT(p)][INBOX(p)] = 1;
    rm[RECV_GRT(p)][MAX(p)] = 1;
    rm[RECV_GRT(p)][STATE(p)] = 1;
  }
}

int* read_matrix(int i) {
  return rm[i];
}

/* Write Matrix
 *          inbox max know state inbox_next
 *  SEND      0    0    0    1    1
 *  RECV_SMALL    1    0    0    0    0
 *  RECV_EQ       1    0    1    1    0
 *  RECV_GREATER  1    1    0    1    0
 */
int wm[NR_ACTIONS * N][NR_VARS * N] = {0};
void set_write_matrix() {
  for(int p = 1; p <= N; p++) {
    wm[SEND(p)][STATE(p)] = 1;
    wm[SEND(p)][INBOX(NEXT(p))] = 1;

    wm[RECV_SMALL(p)][INBOX(p)] = 1;

    wm[RECV_EQ(p)][INBOX(p)] = 1;
    wm[RECV_EQ(p)][KNOW_LEADER(p)] = 1;
    wm[RECV_EQ(p)][STATE(p)] = 1;

    wm[RECV_GRT(p)][INBOX(p)] = 1;
    wm[RECV_GRT(p)][MAX(p)] = 1;
    wm[RECV_GRT(p)][STATE(p)] = 1;
  }
}
```

```

int* write_matrix(int row) {
  return wm[row];
}

int lm[NR_ACTIONS * N + 1][NR_VARS * N] = {0};
void set_label_matrix(int row) {
  for (int p = 1; p <= N; p++) {
    lm[SEND(p)][STATE(p)] = 1;

    lm[RECV_SMALL(p)][STATE(p)] = 1;
    lm[RECV_SMALL(p)][INBOX(p)] = 1;
    lm[RECV_SMALL(p)][MAX(p)] = 1;

    lm[RECV_EQ(p)][STATE(p)] = 1;
    lm[RECV_EQ(p)][INBOX(p)] = 1;
    lm[RECV_EQ(p)][MAX(p)] = 1;

    lm[RECV_GRT(p)][STATE(p)] = 1;
    lm[RECV_GRT(p)][INBOX(p)] = 1;
    lm[RECV_GRT(p)][MAX(p)] = 1;

    lm[LABEL_GOAL][KNOW_LEADER(p)] = 1;
  }
}

int* label_matrix(int row) {
    return lm[row];
}

int gm[NR_ACTIONS * N][NR_ACTIONS * N + 1] = { 0 };
int* guard_matrix(int row) {
  gm[row][row] = 1;

  return gm[row];
}

int dnam[NR_ACTIONS * N][NR_ACTIONS * N] = { 0 };
void set_dna_matrix(int row) {
  for(int p = 1; p <= N; p++) {
    dnam[SEND(p)][RECV_SMALL(NEXT(p))] = 1;
    dnam[SEND(p)][RECV_EQ(NEXT(p))] = 1;
    dnam[SEND(p)][RECV_GRT(NEXT(p))] = 1;

    dnam[RECV_SMALL(NEXT(p))][SEND(p)] = 1;
    dnam[RECV_GRT(NEXT(p))][SEND(p)] = 1;
    dnam[RECV_EQ(NEXT(p))][SEND(p)] = 1;
  }
}

int* do_not_accord_matrix(int row) {
  return dnam[row];
}

int li[NR_ACTIONS * N + 1] = { 0 };
int* label_indices() {
  for (int i  = 0; i < NR_ACTIONS * N + 1; i++) {
    li[i] = i;
  }

  return li;
}

void get_guard_all(void* model, int* src, int g, int* labels) {
  for (int i = 0; i < NR_ACTIONS * N + g; i++) {
    labels[i] = state_label(model, i, src);
  }
}

int state_label(void* model, int label, int* src) {
  if (label == LABEL_GOAL) {
    return label_goal(src);
  }

  for (int p = 1; p <= N; p++) {
    if (label == SEND(p)) {
      return label_send(p, src);
    } else if (label == RECV_SMALL(p)) {
      return label_receive_smaller(model, p, src);
    } else if (label == RECV_EQ(p)) {
      return label_receive_equal(model, p, src);
    } else if (label == RECV_GRT(p)) {
      return label_receive_greater(model, p, src);
    }
  }

  return 0;
}


int label_goal(int* src) {
  int i;

  for(i = 1; i <= N && src[KNOW_LEADER(i)]; i++);
```

```

    return i > N;
}

int label_send(int proc, int* src) {
  return (src[STATE(proc)] == STATE_SEND);
}

int label_receive_smaller(void* m, int proc, int* src) {
  int i, elems[100], max_proc = src[MAX(proc)];
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int size = sset_to_list(inbox_proc, elems, 100);
  for(i = 0; i < size && elems[i] >= max_proc; i++);

  return (src[STATE(proc)] == STATE_RECEIVE &&
      size > 0 && i < size);
}

int label_receive_equal(void* m, int proc, int* src) {
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int max_proc = src[MAX(proc)];

  return (src[STATE(proc)] == STATE_RECEIVE &&
      sset_in(inbox_proc, max_proc));
}

int label_receive_greater(void* m, int proc, int* src) {
  int i, elems[100], max_proc = src[MAX(proc)];
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int size = sset_to_list(inbox_proc, elems, 100);
  for(i = 0; i < size && elems[i] <= max_proc; i++);

  return (src[STATE(proc)] == STATE_RECEIVE &&
      size > 0 && i < size);
}

int ring_send(void* m, int from, int to, int* src, int* dst, int* cpy) {
  int succs = 0;

  if (label_send(from, src)) {//src[STATE(from)] == STATE_SEND) {
    SSET inbox_to = get_sset(m, src, INBOX(to));

    inbox_to = sset_add(inbox_to, src[MAX(from)]);

    dst[INBOX(to)] = set_sset(m, INBOX(to), inbox_to);

    dst[STATE(from)] = STATE_RECEIVE;

    cpy[INBOX(to)] = 0;
    cpy[STATE(from)] = 0;

    succs++;
  }

  return succs;
}

int receive_equal(void* m, int proc, int* src, int* dst, int* cpy) {
  int succs = 0;

  // MAX_i \in INBOX_i
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int max_proc[1] = { src[MAX(proc)] };

//  if (src[STATE(proc)] == STATE_RECEIVE && sset_in(inbox_proc, max_proc[0])) {
  //if (src[STATE(proc)] == STATE_RECEIVE && src[INBOX(proc)] == src[MAX(proc)]) {
  if (label_receive_equal(m, proc, src)) {
    dst[KNOW_LEADER(proc)] = 1;
    // inbox' = inbox - { max }
    sset_difference(&inbox_proc, sset_from_list(max_proc, 1));
    dst[INBOX(proc)] = set_sset(m, INBOX(proc), inbox_proc);
    dst[STATE(proc)] = STATE_SEND;//sset_cardinality(inbox_proc) == 0 ? STATE_SEND :
        STATE_RECEIVE;

    cpy[KNOW_LEADER(proc)] = 0;
    cpy[INBOX(proc)] = 0;
    cpy[STATE(proc)] = 0;

    succs++;
  }

  return succs;
}

int receive_greater(void* m, int proc, int* src, int* dst, int* cpy) {
  int succs = 0;

  int i, elems[100], max_proc = src[MAX(proc)];
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int size = sset_to_list(inbox_proc, elems, 100);
  for(i = 0; i < size && elems[i] <= max_proc; i++);

  //if (src[STATE(proc)] == STATE_RECEIVE && size > 0 && i < size) {
  if(label_receive_greater(m, proc, src)) {
    int new_max_proc[1] = { elems[i] };
    dst[MAX(proc)] = new_max_proc[0];
    sset_difference(&inbox_proc, sset_from_list(new_max_proc, 1));
    dst[STATE(proc)] = STATE_SEND; //sset_cardinality(inbox_proc) == 0 ? STATE_SEND :
        STATE_RECEIVE;
    dst[INBOX(proc)] = set_sset(m, INBOX(proc), inbox_proc);

    cpy[MAX(proc)] = 0;
    cpy[INBOX(proc)] = 0;
    cpy[STATE(proc)] = 0;

    succs++;
  }

  return succs;
}

int receive_smaller(void* m, int proc, int* src, int* dst, int* cpy) {
  int succs = 0;

  int i, elems[100], max_proc = src[MAX(proc)];
  SSET inbox_proc = get_sset(m, src, INBOX(proc));
  int size = sset_to_list(inbox_proc, elems, 100);
  for(i = 0; i < size && elems[i] >= max_proc; i++);

  if (label_receive_smaller(m, proc, src)) {
      //src[STATE(proc)] == STATE_RECEIVE && size > 0 && i < size) {
    int e[1] = { elems[i] };
    SSET rem = sset_from_list(e, 1);

    sset_difference(&inbox_proc, rem);

    dst[INBOX(proc)] = set_sset(m, INBOX(proc), inbox_proc);

    cpy[INBOX(proc)] = 0;

    succs++;
  }

  return succs;
}
```

B.3 DLOPEN IMPLEMENTATION FILE (DLOPEN.C)

```
#include <ltsmin/pins.h>
```

```
#include <ltsmin/pins-util.h>
#include <ltsmin/dlopen-api.h>
#include <ltsmin/ltsmin-standard.h>
#include <ltsmin/lts-type.h>

#include <spec.h>
#include "sSet.h"

#define SIZE 128

// set the name of this PINS plugin
char pins_plugin_name[] = "ring";


static void sl_group (model_t model, sl_group_enum_t group, int *state, int *labels) {
    switch (group) {
        case GB_SL_ALL:
            get_guard_all(model, state, 1, labels);
            return;
        case GB_SL_GUARDS:
            get_guard_all(model, state, 0, labels);
            return;
        default:
            return;
    }
}

void pins_model_init(model_t m) {
    char name[SIZE];

    // create the LTS type LTSmin will generate
    lts_type_t ltstype=lts_type_create();

    // set the length of the state
    lts_type_set_state_length(ltstype, state_length());

    // add an "int" type for a state slot
    int int_type = lts_type_put_type(ltstype, "int", LTStypeSInt32 , NULL);

    // add an "action" type for edge labels
    int action_type = lts_type_put_type(ltstype, "action", LTStypeEnum, NULL);

    // add a "bool" type for state labels
    int bool_type = lts_type_put_type (ltstype, "bool", LTStypeBool, NULL);

    // add an "int" type for a state slot
    int set_type = lts_type_put_type(ltstype, "set", LTStypeChunk, NULL);
```

```

  // set state name & type
  for (int i=1; i <= N; i++) {
  sprintf(name, "inbox_%d", i);
      lts_type_set_state_name(ltstype, INBOX(i), name);
      lts_type_set_state_typeno(ltstype, INBOX(i), set_type);

      sprintf(name, "max_%d", i);
      lts_type_set_state_name(ltstype, MAX(i), name);
      lts_type_set_state_typeno(ltstype, MAX(i), int_type);

      sprintf(name, "know_leader_%d", i);
      lts_type_set_state_name(ltstype, KNOW_LEADER(i), name);
      lts_type_set_state_typeno(ltstype, KNOW_LEADER(i), int_type);

      sprintf(name, "state_%d", i);
      lts_type_set_state_name(ltstype, STATE(i), name);
      lts_type_set_state_typeno(ltstype, STATE(i), int_type);
  }

  // edge label types
  lts_type_set_edge_label_count(ltstype, 1);
  lts_type_set_edge_label_name(ltstype, 0, "action");
  lts_type_set_edge_label_type(ltstype, 0, "action");
  lts_type_set_edge_label_typeno(ltstype, 0, action_type);

  // state label types
  lts_type_set_state_label_count (ltstype, label_count());

for (int i = 1; i <= N; i++) {
  sprintf(name, "label_send_%d", i);
    lts_type_set_state_label_name (ltstype, SEND(i), name);
    lts_type_set_state_label_typeno (ltstype, SEND(i), bool_type);

      sprintf(name, "label_receive_smaller_%d", i);
    lts_type_set_state_label_name (ltstype, RECV_SMALL(i), name);
    lts_type_set_state_label_typeno (ltstype, RECV_SMALL(i), bool_type);

      sprintf(name, "label_receive_equal_%d", i);
    lts_type_set_state_label_name (ltstype, RECV_EQ(i), name);
    lts_type_set_state_label_typeno (ltstype, RECV_EQ(i), bool_type);

      sprintf(name, "label_receive_greater_%d", i);
    lts_type_set_state_label_name (ltstype, RECV_GRT(i), name);
    lts_type_set_state_label_typeno (ltstype, RECV_GRT(i), bool_type);
}

```

```
  lts_type_set_state_label_name (ltstype, LABEL_GOAL, "goal");
  lts_type_set_state_label_typeno (ltstype, LABEL_GOAL, bool_type);

  // done with ltstype
  lts_type_validate(ltstype);

  // make sure to set the lts-type before anything else in the GB
  GBsetLTStype(m, ltstype);

  // setting all values for all non direct types
for (int i = 1; i <= N; i++) {
  sprintf(name, "send_%d", i);
  pins_chunk_put(m, action_type, chunk_str(name));

  sprintf(name, "receive_smaller_%d", i);
  pins_chunk_put(m, action_type, chunk_str(name));

  sprintf(name, "receive_equal_%d", i);
  pins_chunk_put(m, action_type, chunk_str(name));

  sprintf(name, "receive_greater_%d", i);
  pins_chunk_put(m, action_type, chunk_str(name));
}

  // set state variable values for initial state
printf("Loading initial state...\n");
int* initial = initial_state(m);
/*for(int i = 0; i < N; i++) {
  char data[1024];
  int length = sset_serialize(sset_init(), data, 1024);
  initial[INBOX(i)] = pins_chunk_put(m, set_type, chunk_ld(length, data));
} */
  GBsetInitialState(m, initial);

  // set function pointer for the next-state function
  GBsetNextStateLong(m, (next_method_grey_t) next_state);

  // set function pointer for the label evaluation function
  GBsetStateLabelLong(m, (get_label_method_t) state_label);

  // create combined matrix
  matrix_t *cm = malloc(sizeof(matrix_t));
  dm_create(cm, group_count(), state_length());

  // set the read dependency matrix
set_read_matrix();
  matrix_t *rm = malloc(sizeof(matrix_t));
  dm_create(rm, group_count(), state_length());
  for (int i = 0; i < group_count(); i++) {
  int* aux = read_matrix(i);
      for (int j = 0; j < state_length(); j++) {
          if (aux[j]) {
              dm_set(cm, i, j);
              dm_set(rm, i, j);
          }
      }
  }
  GBsetDMInfoRead(m, rm);

  // set the write dependency matrix
set_write_matrix();
  matrix_t *wm = malloc(sizeof(matrix_t));
  dm_create(wm, group_count(), state_length());
  for (int i = 0; i < group_count(); i++) {
  int* aux = write_matrix(i);
      for (int j = 0; j < state_length(); j++) {
          if (aux[j]) {
              dm_set(cm, i, j);
              dm_set(wm, i, j);
          }
      }
  }
  GBsetDMInfoMustWrite(m, wm);

  // set the combined matrix
  GBsetDMInfo(m, cm);

  // set the label dependency matrix
set_label_matrix();
  matrix_t *lm = malloc(sizeof(matrix_t));
  dm_create(lm, label_count(), state_length());
  for (int i = 0; i < label_count(); i++) {
  int* aux = label_matrix(i);
      for (int j = 0; j < state_length(); j++) {
          if (aux[j]) dm_set(lm, i, j);
      }
  }
  GBsetStateLabelInfo(m, lm);

  // set the do not accord dependency matrix
set_dna_matrix();
  matrix_t *dnam = malloc(sizeof(matrix_t));
  dm_create(dnam, group_count(), group_count());
  for (int i = 0; i < group_count(); i++) {
    int* aux = do_not_accord_matrix(i);
        for (int j = 0; j < group_count(); j++) {
            if (aux[j]) dm_set(dnam, i, j);
        }
    }
  GBsetDoNotAccordInfo(m, dnam);


  guard_t** guards = malloc(group_count() * sizeof(guard_t*));
  for(int i = 0; i < group_count(); i++) {
    guards[i] = malloc(sizeof(guard_t) + sizeof(int));
    guards[i]->count = 1;
    guards[i]->guard[0] = i;
  }
  GBsetGuardsInfo(m, guards);

  // set group info
    sl_group_t* group_all = malloc(sizeof(sl_group_t) + label_count() * sizeof(int));
    group_all->count = label_count();
  memcpy(group_all->sl_idx, label_indices(), group_all->count * sizeof(int));

    sl_group_t* group_guards = malloc(sizeof(sl_group_t) + guard_count() * sizeof(int));
    group_guards->count = guard_count();
  memcpy(group_guards->sl_idx, label_indices(), group_guards->count * sizeof(int));

    GBsetStateLabelGroupInfo(m, GB_SL_ALL, group_all);
    GBsetStateLabelGroupInfo(m, GB_SL_GUARDS, group_guards);
    GBsetStateLabelsGroup(m, sl_group);
}
```

C

TAINO RING LEADER ELECTION

```
enum State { Send, Receive }

var sig elected {}

sig Int {}

sig Node {
   var max : one Int,
   succ : one Node,
   var inbox : set Int,
   var state : one State
}

event send[n : Node] {
  n.state = send

  n.succ.inbox' = n.succ.inbox + n.max
  n.state' = receive
}

event receiveSmall[n : Node, m : Int] {
  n.state = Receive
  m in n.inbox
  lt[m, n.max]

  n.inbox' = n.inbox - m
}

event receiveGreater[n : Node, m : Int] {
  n.state = Receive
  m in n.inbox
  gt[m, n.max]

  n.max' = m
  n.inbox' = n.inbox - m
  n.state' = Send
}

event receiveEqual[n : Node, m : Int] {
  n.state = Receive
  m in n.inbox
  n.max = m

  elected' = elected + m
  n.inbox' = n.inbox - m
  n.state' = Send
}

init {
  no elected
  Node$1.max = 1
  Node$1.succ = Node$2
  no Node$1.inbox
  Node$1.state = Send

  Node$2.max = 2
  Node$2.succ = Node$3
  no Node$2.inbox
  Node$2.state = Send

  Node$3.max = 3
  Node$3.succ = Node$4
  no Node$3.inbox
  Node$3.state = Send

  Node$4.succ = Node$5
  no Node$4.inbox
  Node$4.state = Send
  Node$4.max = 4

} for 4 Node, 4 Int

pred only_one_elected {
  one elected
}

check {
  eventually always only_one_elected
}
```