Verifying Properties of State-Based Models using Constraint Programming

Victoria Johnson[1], Pedro Ribeiro[2], Simon Foster[2], Peter Nightingale[2], and Felix Ulrich-Oltean[2]

[1] University of Sheffield, UK
v.johnson@sheffield.ac.uk
[2] Department of Computer Science, University of York, UK
{pedro.ribeiro,simon.foster,
peter.nightingale,felix.ulrich-oltean}@york.ac.uk

Abstract. We explore the application of Constraint Programming (CP) tools to modelling state-based systems and verifying their properties. This includes finding execution traces leading to a particular state, and proving deadlock-freedom up to a given bound on the number of transitions. We present three distinct case studies. The first formulates a railway signal in the Essence CP modelling language, demonstrating use of Essence types and operators to model states, transitions, and invariants, in a system with a single finite-state automaton. The second case study is based on Dining Philosophers, and demonstrates effective CP modelling of a system with a large number of automata, synchronised on transitions. The third case study is part of the Alpha Algorithm, an example from swarm robotics. It introduces a clock, and has transitions with guards that refer to the clock. It also has triggers, representing sensor inputs, and non-deterministic waits, demonstrating that these concepts can be represented in a CP model. Finally we demonstrate that the CP approach is complementary to a model checking approach using FDR4. In many cases the CP approach can scale substantially better than the model checker, despite the CP toolchain being general-purpose, i.e. not explicitly designed for verifying properties of state-based models.

1 Introduction

Automating the verification of state-based systems in notations like Z [33,34] and RoboChart [22] state machines requires that we can exhaustively analyse the state spaces for scenarios that violate the safety requirements. This can be achieved using model checkers like FDR [11] and SPIN [15], which enumerate a labelled transition system, and then use this to search for properties of interest. Model checking is fully automated, but suffers from the state-explosion problem. In contrast, theorem proving also allows verification that overcomes state-explosion through an implicit encoding of the transition relation using symbolic logic, but usually requires manual intervention, such as finding invariants. A middle alternative is the use of Contraint Programming (CP). Like theorem

proving, CP uses symbolic logic, and so allows a large number of states to be considered efficiently without enumerating all the states, but at the same time has a high level of automation as in model checking.

In this paper, we consider how CP can be harnessed as a method for verifying state-based languages, such as RoboChart state machines. Since constraint solvers avoid searching every possible state with efficient algorithms, our hypothesis is that we can use them as a more efficient complementary technique to model checking. We therefore evaluate CP as a technique for modelling and verifying a variety of state-based languages, encompassing diverse paradigms such as non-determinism, concurrency, and real-time.

We encode the transition system of a state-based model as a matrix in the Essence language [10], and then formulate a property of interest as a constraint problem. In particular, we can use CP to search for deadlocking states by formulating the existence of a deadlock as a constraint problem, and similarly for the violation of an invariant. If a deadlocking or erroneous state exists then the Savile Row tool can locate a trace of a maximum given length that shows this.

We apply this technique to three case studies: a railway signal [20], the dining philosophers problem, and the Alpha Algorithm [5] for swarms. The final case study uses the RoboChart language, and applies a systematic translation including timed transitions. For the latter two case studies, we perform a side-by-side comparison with FDR, and evaluate our findings. Our early results demonstrate that, subject to a time horizon on the model, CP can achieve greater scalability than model checking, and so provides a useful complementary technique.

The structure of our paper in as follows. In Section 2 we introduce the background for our paper: Constraint Programming and RoboChart. In Section 3, we introduce our approach by encoding a Dwarf railway signal model in the Essence language. In Section 4 we present a systematic encoding from state machines into Essence, which is used by the two case studies. In Section 5 we model the dining philosophers problem. In Section 6 we introduce the encoding of the Alpha Algorithm. In Section 7 we outline the results of our experiments. In Section 8 we highlight related work, and in Section 9 we conclude.

2 Background

The earliest forms of state machines appear in the foundational theories of computation, namely in the form of Moore [23] and Mealy [21] machines that make an explicit distinction between inputs and outputs. Harel's statecharts [14] expanded on those ideas by associating trigger events, conditions, and actions with transitions. A condition is specified by a predicate, over some state variables, that guards a transition's applicability. Other related formalisms, such as, Extended Finite State Machines (EFSMs) [7] and UML support a similar style of specification. This is in contrast with the simple notion of events between edges of a graph, as captured by Labelled Transition Systems (LTS), for example, at the core of operational [29] accounts of reactive systems.

RoboChart One of our case studies is based on a RoboChart [22] model. RoboChart is a Domain-Specific Language (DSL) for the design of timed reactive control software. It supports a model-driven engineering approach to development, focusing on high-level system design. Its component model provides for the definition of self-contained behavioural components based on state machines that are platform-independent. The core notation is based on that of UML statecharts, however, it avoids some of the concepts that can make formal reasoning challenging, such as inter-level transitions. Distinctively, in RoboChart, state machines provide primitives for specifying time budgets and deadlines and have a formal semantics given in *tock*-CSP [2,29], a discrete-timed process algebra with support for verification using the model checker FDR [11].

Constraint Programming Constraint Programming (CP) [30] is a paradigm for representing and solving combinatorial problems. In CP, a problem is modelled (represented) as a set of decision variables, a set of constraints on subsets of the decision variables, and an optional optimisation function. CP draws on techniques from artificial intelligence and operations research to develop effective general-purpose solvers for combinatorial problems, as well as tools that support the modelling process. CP modelling tools typically read in a modelling language and reformulate it to produce output for a chosen solver or class of solvers. Modelling tools automate the often tedious process of translating constraints for a particular solver, while also reformulating the model to improve solver performance. Prominent examples include OPL [32] and MiniZinc [25].

In this work we use two modelling languages: Essence [10] and the simpler lower-level language Essence Prime. Essence includes several types of decision variables that Essence Prime does not (sets, multisets, functions, partitions, relations, and variable-length sequences). They both have decision variable matrices (containing elements of any type in the language), integers, and Booleans. We use Essence when we need set variables and Essence Prime otherwise. When using Essence, we use the automated modelling tool Conjure [1] 2.6.0 to refine it to Essence Prime. In both cases we use Savile Row [26] 1.11.1 to translate Essence Prime into SAT (using default SAT encoding settings) and solve with the SAT solver Kissat [3] 3.1.1 or Google OR-Tools CP-SAT [27] 9.11.

3 A State-Based Model in Essence: Dwarf Signal

In this section we give an example to illustrate the use of a CP modelling language to verify properties of a state-based model. A Dwarf Signal is a type of railway signal used in places where track-side space is limited. It has three lamps, as shown in Figure 1. Their implementation is based on the old semaphore signalling system in which a bar (arm) was raised and lowered into vertical, horizontal and diagonal positions. Clearly, railway signals need to be safe and reliable in their implementation, and so the Dwarf Signal has been given a precise specification [20]. The signal has three lamps, which can be lit or unlit, and are displayed in different configurations to give instructions to train drivers. The

Fig. 1: A Dwarf Signal

three lamps are named $L1 - L3$, as illustrated, and a signal configuration is a subset of $\mathit{Signal} = \mathbb{P}\{L1, L2, L3\}$. Different configurations of a signal can be written using set notation, for instance the signal in Figure 1 is in $\{L1, L2\}$, which means *stop*. The signal has a total of four "proper states," which are the well-defined commands a signal can convey to a driver:

$$\mathit{dark} = \{\} \quad \mathit{stop} = \{L1, L2\} \quad \mathit{warning} = \{L1, L3\} \quad \mathit{drive} = \{L2, L3\}$$

The controller of the dwarf signal transitions between proper states by lighting and extinguishing the three lamps one at a time. The state variables include

- $\mathit{currentState} : \mathit{Signal}$, which gives the current configuration;
- $\mathit{desiredProperState} : \{\mathit{dark}, \mathit{stop}, \mathit{warning}, \mathit{drive}\}$, with the next proper state;
- $\mathit{turnOn}, \mathit{turnOff} : \mathbb{P}\{L1, L2, L3\}$, which gives the lights that need to be respectively lit or extinguished to reach the desired proper state.

The system is specified with three operations: $\mathit{TurnOn}(l)$, which turns a lamp $l \in \{L1, L2, L3\}$ on, $\mathit{TurnOff}(l)$, which turns a lamp off, and $\mathit{SetNewProperState}(s)$, which requests that the signal transitions to a new proper state. We can encode this system in Essence as follows.

Decision Variables As described in Section 2, CP modelling languages provide decision variables of various types including integers, booleans, sets of integers, and in some cases more complex types such as relations and functions. However they do not provide a built-in way to define states with transitions from one state to another. Therefore states and transitions must be modelled in the language. The state of the Dwarf Signal can simply be a set of integers drawn from $\{1 \dots 3\}$. Its evolution over time can be represented as a time-indexed array of sets of integers, as follows. The `horizon` is the final time step.

```
find currentState : matrix[int(1..horizon)] of set of int(1..3)
```

In the declaration above, the `find` keyword indicates that `currentState` is a matrix of decision variables, whose values are unknown in advance and will be decided by the constraint solver.

We have chosen to use the Essence language in this case because it supports the required types of decision variables. Alongside `currentState` we define the following time-indexed arrays representing traces of the other state variables.

```
find lastProperState : matrix[int(1..horizon)]
    of set of int(1..3)
find turnOff : matrix[int(1..horizon)] of set of int(1..3)
find turnOn : matrix[int(1..horizon)] of set of int(1..3)
find desiredProperState : matrix[int(1..horizon)]
    of set of int(1..3)
```

We now need to model the three operations of the Dwarf Signal. For each timestep except the last, we define a decision variable for the chosen operation and two other variables representing operation parameters (which may or may not be required, depending on the chosen operation). The operations are numbered $0, 1, 2$, corresponding to *TurnOff*, *TurnOn* and *SetNewProperState*.

```
find action : matrix [ int(1..horizon-1) ] of int(0..2)
find actionLampId : matrix [ int(1..horizon-1) ] of int(1..3)
find newDesiredState : matrix [ int(1..horizon-1) ]
    of set of int(1..3)
```

Constraints Having defined the decision variables, we need to connect one timestep to the next with constraints. Essence includes common operators that can be used to write constraints for each of its abstract types (set, multiset, variable-length sequence, function, relation, partition). Here, we are using sets, and Essence provides union, intersection, set difference, and cardinality among others. The following constraints implement the *TurnOn* operation (action 1).

```
1 forAll t : int(1..horizon-1).
2   action[t]=1 -> and([
3     actionLampId[t] in turnOn[t],
4     lastProperState[t+1]=lastProperState[t],
5     turnOff[t+1]=turnOff[t],
6     turnOn[t+1]=turnOn[t]-{actionLampId[t]},
7     currentState[t+1]=currentState[t] union {actionLampId[t]},
8     desiredProperState[t+1]=desiredProperState[t]
9     ]),
```

The first and second lines ensure that the constraints are applied for each timestep where operation 1 is chosen. Line 3 represents a precondition, which states that the lamp chosen to turn on must be in the set `turnOn`. Lines 4-8 update (or carry forward unchanged) the value in each state matrix from timestep `t` to `t+1`. The other two actions are implemented similarly.

The initial state of the dwarf signal is set to *stop* as follows.

```
currentState[1]={1,2},          lastProperState[1]={1,2},
turnOff[1]={},                  turnOn[1]={},
desiredProperState[1]={1,2},
```

Finally, decision variables representing action parameters (`actionLampId` and `newDesiredState`) are set to a default value when they are not needed. For the *turnOn* and *turnOff* operations, only `actionLampId` is needed, while the *setNewProperState* action requires only `newDesiredState`.

Invariants We can check whether it is possible to violate an invariant within a given time horizon. The simple **NeverShowAll** invariant states that we should never have all lamps on simultaneously, formally $currentState \neq \{L1, L2, L3\}$. We write the negation (`!`) of the invariant in order to find a counterexample trace, if one exists. Representing **NeverShowAll**, the following constraint states that it is *not* the case that the invariant holds for all time steps. In other words, there must exist (in a counterexample trace) a timestep when the invariant is violated. The constraint solver will either find a trace or report that none exist.

```
!forAll t: int(1..horizon) . currentState[t]!={1,2,3},
```

Solving the model with this constraint and a horizon of 5 produced the following trace for `currentState`, indicating an invariant violation: `[{1, 2}, {1, 2}, {1, 2, 3}, {1, 3}, {1, 3}; int(1..5)]`. In the third state all three lights are on, which is due to insufficient preconditions on the *TurnOn* operation. A solution is to extinguish the required lamps prior to lighting others.

Invariants can refer to multiple time steps, and any of the state variables. The **MaxOneLampChange** invariant requires that at most one lamp changes state from one timestep to the next. Using the set operators available in Essence, **MaxOneLampChange** can be expressed as follows:

```
!forAll t: int(2..horizon).
  exists l : int(1..3). (
    currentState[t] = currentState[t-1] union {l} \/
    currentState[t-1] = currentState[t] union {l} \/
    currentState[t-1]=currentState[t]
  ),
```

Once again the invariant is negated: the constraint solver will search for a counterexample trace where the invariant is violated at any timestep `t` within the horizon. Solving the model with **MaxOneLampChange** and a horizon of 100 produced no solutions (**MaxOneLampChange** was not disproven).

The **ForbidStopToDrive** invariant requires that the signal never transitions from *stop* to *drive* without another proper state in between. The invariant does not mention the current state, but instead uses the last proper state, and the desired proper state at timestep `t`. It is modelled as follows.

```
!forAll t: int(1..horizon) .
  lastProperState[t]=stop -> desiredProperState[t]!=drive,
```

Solving the model with **ForbidStopToDrive** and a horizon of 5 produces the following trace for `currentState`, showing an invariant violation: `[{1, 2}, {1, 2}, {1, 2, 3}, {2, 3}, {2, 3}; int(1..5)]`. There is a transition straight from *stop* to *drive* due to a missing precondition on *SetNewProperState*.

In this extended example we have shown that a simple state-based model with set variables can be represented naturally in Essence, and that a CP toolchain can be straightforwardly used to check invariants. Counterexample traces produced by the constraint solver can then be used to improve the system so that the desired invariants hold. In the next section we move from a simple state-based model to state machines with guarded transitions and other features.

4 Encoding State Machines into CP

In this section we describe how we encode finite state automata into CP in a general way, while also prioritising constraint solver performance. We encode both states and transitions as integers, and implement transitions using a type of constraint that allows us to list the transitions explicitly. The approach taken in this section is somewhat different from the Dwarf Signal example in Section 3, where the transitions (turning a lamp on or off, setting a new desired state) correspond to common set operations. In general we cannot assume that state-based models will conform to the operations available in a high-level modelling language such as Essence.

Following Quimper and Walsh's decomposition of the regular language constraint [28], we model state machines in CP using integer decision variables for both states and transitions. State variables, denoted `S[p,k]` for automaton `p` at time step `k`, and defined within a set of possible states, encode the state of the automaton at time step `k`. Transition variables `T[p,k]` encode the transition from step `k` to `k+1`. The validity of a transition is enforced using a table constraint. Table constraints are used to ensure that for every timestep `k`, the tuple `(S[p,k], T[p,k], S[p,k+1])` corresponds to a row in the transition table provided by the user. Finally, to encode the initial state, we assign `S[p,1]` to a constant for each automaton `p`. The model fragment below shows the CP model of the Philosopher state machine from Section 5 (where `philTT` is the transition table: one row for each transition specifying the origin state, transition number, and subsequent state).

```
find Sp: matrix[int(1..nP), int(1..horizon)] of int(1..4)
find Tp: matrix[int(1..nP), int(1..horizon - 1)] of int(1..8)
....
forAll k: int(1..horizon - 1). forAll p: int(1..nP).
    table([Sp[p, k], Tp[p, k], Sp[p, k + 1]], philTT),
forAll i: int(1..nP). Sp[i, 1] = 1,
```

4.1 Modelling Guarded Transitions

In some kinds of state models, transitions may be guarded by arbitrarily complex expressions referencing external timers, states, variables, or other elements. To encode this in CP, the conditions for a transition are separated into two boolean matrices: `guard` and `trigger`. `guard[p,k,t]` represents the internal conditions that must be true for transition `t` to be available. `trigger[p,k,t]` represents external events or inputs required for the transition. For a transition `t` to be taken at time step `k`, both the guard and trigger must be true:

```
T[p,k]=t -> (guard[p,k,t] /\ trigger[p,k,t]),
```

4.2 Maximal Progress Assumption and Time Transitions

For both our case studies we have *time transitions* (i.e. transitions that do not change the state of the automaton but allow one unit of time to pass). The maximal progress assumption requires that an automaton does not idle if it can act at any given timestep, i.e. it cannot use a time transition if any other transition is possible. We add constraints to enforce the maximal progress assumption: if a transition `t2` is available (its guard and trigger are both true), the time transition `t1` is disallowed. Here, the set of transitions that are time transitions are called `TimeT`, and the set of all other transitions are called `NonTimeT`.

4.3 Modelling Properties to Check

It is possible to represent various interesting properties in CP in a general way, allowing us to check them (to the horizon) using CP tools. In this section we give representations of deadlock freedom, reachability, and divergence freedom.

The simplest form of deadlock occurs if the system reaches a state where the only available transitions (for which the guard is *true*) for all automata are time transitions. It suffices to constrain a subset of the guards to be *false* at the horizon. An example can be found in the Dining Philosophers model (Section 5).

For state-based models that contain clocks, guards can become *true* simply by time passing without any other changes to the state. To model this scenario we define a *deadlock horizon* `h2`, and we consider the system to be deadlocked if all guards are *false* (except the guards of time transitions) for all automata, for any sequence of `h2` consecutive time steps. An example of this kind of deadlock detection can be found in the the Alpha Algorithm model in Section 6.

State reachability ensures that specific states can be visited within the horizon. This can be expressed as follows (where `i` is a state and `p` is a finite state automaton): `exists k: int(1..horizon). S[p,k]=i`

Divergence freedom in general ensures the system does not enter an infinite loop of internal actions without producing observable outputs. This could be, for example, remaining within a subset of the states. Given a pair of states `s1` and `s2`, we can test whether the system can visit `s1`, then `s2`, then `s1` again as follows, for some automaton `p`.

```
exists k,a,b : int(1..horizon).
    S[p,k]=s1 /\ S[p,k+a]=s2 /\ S[p,k+a+b]=s1,
```

5 Case study 1: Dining Philosophers

The Dining Philosophers problem deals with synchronisation of resource allocation and management. In the problem, there are `nP` philosophers and forks, arranged alternating in a circle. The states of the philosophers are thinking, hungry (and searching for forks), and eating. Each philosopher is only permitted to eat if they have picked up the forks on both their left and right sides. For this particular implementation, the philosophers have four states: (1) thinking; (2)

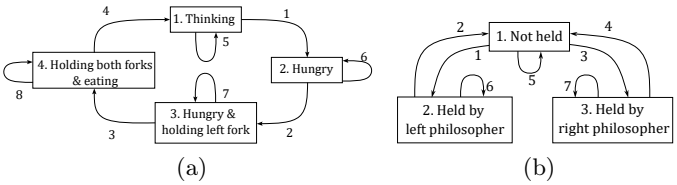


Fig. 2: State diagrams for (a) the philosophers, and (b) the forks.

hungry and holding no forks; (3) hungry and holding the left fork; and (4) holding both forks and eating. Transitions must always happen "forwards" – philosophers may proceed from state `i` to `i+1` but never `i` to `i-1`. Transition from state 4 to state 1 is permitted as well. All states have a time transition. A philosopher may only pick up the forks on either side if they not already held by another philosopher. The state diagram for the philosophers is shown in Figure 2a.

Similarly, the forks have three states: (1) not held; (2) held by the philosopher on the left; (3) held by the philosopher on the right. The forks are able to transition from (1) to (2), and (1) to (3) freely, but must transition through state 1 when being transferred from one philosopher to the other. The state diagram for the forks is shown in Figure 2b. The state decision variables for the philosophers and forks are denoted `Sp` and `Sf` respectively, while the transition decision variables are denoted `Tp` and `Tf`. In addition we have the horizon and the number of philosophers/forks as parameters.

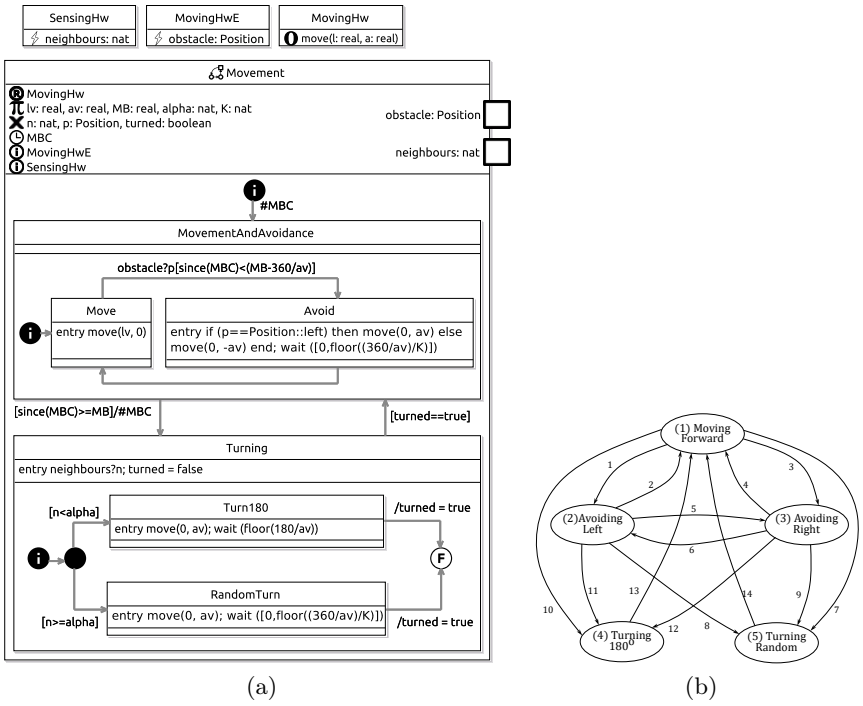
The automata are synchronised on transitions, and we express this with the following constraints, where `p_right = (p - 2)%nP + 1`:

```
Tp[p,k]=2 <-> Tf[p,k]=3,          Tp[p,k]=3 <-> Tf[p_right,k]=1,
Tp[p,k]=4 <-> Tf[p_right,k]=2,    Tp[p,k]=4 <-> Tf[p,k]=4
```

For example, philosopher `p`'s transition 2 (pick up the fork on the left) can only happen if the fork on the left is being picked up by the philosopher to its right (the first constraint above). The initial conditions are that all philosophers are in state 1 (thinking), and all forks are in state 1 (not held).

We use deadlock-freedom as an example of a property that would be desirable to verify. Deadlock occurs if each philosopher holds the fork to their left simultaneously, preventing any other philosopher from making a transition (other than time transitions). The deadlock detection method discussed in Section 4.3 is used, with guards indicating if a transition is possible for both the philosophers and the forks. If the system is deadlocked, all guards of transitions that are not time transitions (`philNotTimeT` and `forkNotTimeT`) at the horizon are *false*, as described in Section 4.3. We have the following constraint expressing deadlock for all philosophers and forks simultaneously.

```
forAll p: int(1..nP).
    (forAll t : philNonTimeT. !guard_p[p, horizon-1, t]) /\
    (forAll t : forkNonTimeT. !guard_f[p, horizon-1, t]),
```



(a) (b)

Fig. 3: (a) RoboChart state machine Movement; (b) The finite state diagram representing Movement. The time transitions are not included here for brevity.

6 Case study 2: Alpha Algorithm

The Alpha Algorithm [5] is a swarm robotics algorithm designed to disperse robots when the swarm density becomes too high. In this case study, a single robot is modelled within a swarm. There are two behaviours of a robot acting under the Alpha Algorithm, which we model here via two interacting RoboChart state machines: a Movement machine, that determines the physical movement of the robot, such as moving, turning, and reacting to the density of the swarm; and a Communication machine (omitted), that determines the communications of a robot with its neighbours. In this case study, we consider only the Movement machine (), reproduced in Figure 3a, as we explain next.

The contents in the top box of Movement in Figure 3a define the machine's context. It (Ⓡ) requires an interface MovingHW that defines an operation (Ⓞ) of the robot called move with two parameters of type real for linear and angular velocity, and uses (ⓘ) the interfaces MovingHwE and SensingHw, that define (typed) input events (↯) obstacle, of an enumerated type Position with possible values left or right, and neighbours of the natural type. Using these events the machine can receive information about the relative position of obstacles and the

number of neighbouring robots currently detected. In addition, we declare (π) constants, (✕) variables, and a (◷) clock MBC.

The constants lv and av are used for setting the linear and angular velocity, while MB is used to decide how long to perform MovementAndAvoidance, alpha is the threshold of the alpha algorithm, and K is a constant that controls the time interval allowed for turns. The variables n and p are used to store values received via the events neighbours and obstacle, respectively, while turned is an auxiliary variable that aids with the control flow specification for state Turning.

The control flow of the machine Movement starts in the initial junction (indicated by ⓘ), whose outgoing transition (#) resets the clock MBC and identifies the starting state of the control flow. In Movement, that state is the composite state MovementAndAvoidance, whose control flow also starts from its initial junction and therefore state Move. In Move, there is an entry action, where the operation move is called with values lv and 0. Here, we assume that move is an operation provided by the robot's embedded software that is always available, and so the call takes no time. After completing the entry action, state Move and MovementAndAvoidance become active, so any outgoing transitions, if enabled, can interrupt their execution. For example, the transition from Move to Avoid can be triggered by the event obstacle when the condition since(MBC)<(MB-360/av) is true, where since(MBC) is the time elapsed since the last reset of clock MBC. If triggered, the value communicated via obstacle is assigned to the variable p, leading to state Avoid. In Avoid, there is an entry action, where if the obstacle is positioned on the left, then move is called with zero linear velocity but positive angular velocity av, and otherwise negative angular velocity -av, and afterwards there is a non-deterministic delay (wait) between 0 and floor((360/av)/K) time units. The transition back to Move has no trigger or condition, so it must be taken as soon as Avoid finishes its entry action to ensure maximal progress.

Similarly, in MovementAndAvoidance, after entering states Move or Avoid, the transition to Turning, guarded by since(MBC)>=MB, must be taken as soon as it is enabled to ensure maximal progress. If it is enabled at the same time as a transition between Move and Avoid, then the choice over which transition is taken is nondeterministic. So, in Turning we use the auxiliary variable turned to eliminate this kind of nondeterminism. In its entry action, we have, first of all, an input communication on neighbours and then the assignment of false to turned. In the control flow of Turning, there is a junction, with mutually exclusive conditions: if n<alpha then control transitions to Turn180, and otherwise to RandomTurn. The behaviour of their entry actions is similar in that both call move(0,av), however, the subsequent waiting period is floor(180/av) in the first case, and otherwise is a nondeterministic delay between 0 and floor((360/av)/K) time units in RandomTurn, similarly to that in state Avoid. Once the entry action completes, control transitions to a final state, with the action on the transitions setting turned to true, thus enabling the transition from Turning to MovementAndAvoidance. This concludes our discussion of the Movement state machine.

Figure 3b shows a flattened version of Movement as a state transition diagram used as starting point for our encoding in CP. Figure 3b was modelled

in Essence Prime using the encoding framework from Section 4. The implementation includes user-defined parameters that define the physical limits and thresholds of the system, and decision variables that track the robot's state from the first timestep to the horizon. Parameters include the number of simulation timesteps (`horizon`), `alpha` and `av` (both described above).

The main decision variables are: the sequence of states (`Smov`), transitions (`Tmov`), and `guard` and `trigger` for each transition at each step (as described in Section 4). The maximal progress assumption is included, as are time transitions. To encode the timed elements of the Alpha Algorithm, we define variables to track MBC (named `MBC`), the time taken at each simulation step (`timings`), and the total elapsed time (`cumulTime`) for each step. The value of `MBC` is carried forward unchanged at each step unless the clock is reset (in which case `MBC` is constrained to be equal to `cumulTime`). The `timings` variables are constrained according to the wait instructions in the RoboChart model (defaulting to 1 if there is no wait for the current state). The `cumulTime` variables simply accumulate `timings` for each step.

We also define decision variables for the random turn duration `lambda` (with domain $\{0 \dots 72\}$ in our experiments) and the number of robots detected in the vicinity (`nDet` with domain $\{0, \texttt{alpha} - 1, \texttt{alpha}, \texttt{alpha} + 1\}$) at each step. If the robot is in a random turning state, the turning time is proportional to `lambda`. However, if the robot is performing a 180-degree turn, the duration is fixed based on the angular velocity. The value of `nDet` affects the guards on some of the transitions, as described below. Finally we define `sensor` $\in \{-1, 0, 1\}$ for each step, corresponding to obstacle with values Position::left, none, or Position::right in the RoboChart model. The `sensor` variables are used in triggers for transitions that enter the Avoid states.

For each simulation step `k` and each transition `t`, we translate the guards and triggers in RoboChart to constraints on `guard[k,t]` and `trigger[k,t]`. As an example, consider transition 1 (from state (1) Moving Forward to (2) Avoiding Left). Transition 1 is available when the automaton is in state 1, and the elapsed time since MBC was last reset is less than a constant. Transition 1 is triggered if the sensor returns a value, and the value is 1 (corresponding to Position::right). The guard and trigger at step `k` are constrained as follows:

```
guard[k, 1] <-> (Smov[k]=1 /\ (cumulTime[k]-MBC[k] < MB-360/av))
trigger[k, 1] <-> (sensor[k] = 1)
```

Note how the external event (a sensor reading) is captured in the trigger, and internal conditions in the guard.

We model timed deadlock as follows (where `time_t` is the set of time transitions for the automaton), following the approach described in Section 4. Recall that `h2` is a parameter: the deadlock horizon. For the system to be considered deadlocked, a set of guards must be false for `h2` consecutive steps.

```
exists t: int(1..horizon-h2+1). forAll transition: int(1..nTm).
    (!(transition in time_t)) ->
        forAll t2 : int(t..t+h2-1). !guard[t2, transition],
```

	FDR4			CP Toolchain	
Phil.	States	Transitions	Time (s)	SR Time (s)	Solv Time (s)
2	17	59	1.27	0.20	0.00
4	389	2,533	1.34	0.30	0.00
6	9,656	92,527	2.11	0.42	0.01
8	236,996	3,000,560	2.75	0.59	0.01
10	6,067,403	95,176,648	4.05	0.79	0.02
11	30,198,167	519,705,897	8.77	0.90	0.03
12	150,317,646	2,816,473,446	33.83	1.06	0.01
13	761,791,638	15,418,186,708	191.86	1.10	0.04
14	3,822,052,236	83,132,411,124	1218.44	1.17	0.04
15	19,203,016,180	446,657,002,703	7510.32	1.36	0.05

Table 1: Results of verifying deadlock freedom of dining philosophers with FDR4, without using partial order reduction, and CP. Complexity is reported in terms of states and transitions visited, and the time taken to find a counter-example. CP time consists of SR (Savile Row) time and time spent by the backend solver.

7 Experimental Results

We compare the performance of our approach to verification, based on our CP encoding, versus model checking with FDR4 [11] of comparable CSP models for our case studies. For the Alpha Algorithm, its *tock*-CSP semantics can be automatically calculated using RoboTool[3], while for the Dining Philosophers we construct an untimed CSP model that encodes the state diagram in Fig. 2.

Experiments were carried out on a server with two AMD EPYC 7501 processors and 2 TB of RAM. In some cases experiments were parallelised but processes were never contending for CPU cores or RAM. Model and parameter files are available: `https://github.com/pwn1/abz26-experiments`.

FDR4 allows verifying properties via refinement [24]. Briefly, FDR calculates the operational semantics of a CSP process explicitly as a LTS. So, an untimed deadlock manifests as a state in the LTS that refuses every possible interaction, whereas a timed deadlock occurs when, despite maximal progress, *tock* becomes the only possible interaction indefinitely. In contrast with CP, FDR4 does not require a time horizon, exploring the LTS in a breadth-first-search manner.

The results of checking the Dining Philosophers model for deadlocks are summarised in Tables 1 and 2. Table 1 shows the results for FDR4 using its default settings, with the number of states and transitions visited and the time taken, and for the CP toolchain, as the number of philosophers grows. We used the `-O0` option for Savile Row 1.11.1 to reduce the time spent performing presolving optimisations. The backend solver was Kissat 3.1.1, and the horizon was 1.5 times the number of philosophers. We observe that the model is not deadlock

[3] `https://robostar.cs.york.ac.uk/robotool`

	FDR4			CP Toolchain	
Phil.	States	Transitions	Time (s)	SR Time (s)	Solv Time (s)
10	37	81	0.95	0.79	0.02
20	85	194	1.30	2.10	0.06
40	165	380	2.47	6.87	0.17
80	290	659	6.71	22.59	0.89
100	404	902	10.35	35.47	1.37
200	803	1,782	84.12	162.87	6.16
400	1,631	3,684	1041.58	825.17	28.19
800	3,305	7,567	14,940.89	4605.43	134.22
1000	4,071	9,153	35,734.29	7953.44	226.84

Table 2: Results of verifying deadlock freedom of dining philosophers with FDR4 using partial order reduction and CP. Complexity is reported in terms of states and transitions visited, and the time taken to find a counter-example.

free. FDR4 shows exponential growth in time, whereas the CP approach is nearly constant up to 15 philosophers.

In Table 2 we show another set of results for the Dining Philosophers model, but this time using FDR4's support for partial order reduction [12] that can tackle a larger number of philosophers. With up to 300 philosophers, FDR4 completes the check faster than CP, with smaller complexity than before (Table 1), but for larger values the verification time grows faster than for CP, with FDR spending most of the time on computing the reduction prior to verification.

For the Alpha Algorithm, we fix the values of the RoboChart model constants as $\mathsf{av} = 5$, $\mathsf{lv} = 1$, $\mathsf{alpha} = 8$ and $\mathsf{K} = 1$, and define the discrete domains of the model types as follows: $\mathsf{nat} = \{0, \mathsf{alpha} - 1, \mathsf{alpha}, \mathsf{alpha} + 1\}$, $\mathsf{real} = \{0, -\mathsf{av}, \mathsf{av}, -\mathsf{lv}, \mathsf{lv}\}$, so that all states in Turning are reachable. Moreover, we introduce a constant D, that we vary for our experiments, and define MB as $(360/\mathsf{av}) + \mathsf{D}$, using integer division. We also let clock MBC take discrete values from 0 to MB+1, inclusive, ensuring that state Turning is reachable.

For the CP model of the Alpha Algorithm, we set parameters to the same values as in the RoboChart model. We also set `horizon` to be 110, and `h2` (the deadlock horizon) to 100 (i.e. to be considered deadlocked, the relevant guards must be false for 100 consecutive steps, taking 100 time units). We also include a parameter `D` that takes the same value and has the same effect as D in the RoboChart model. In this case we use Savile Row 1.11.1 with default settings, and Google OR-Tools 9.11 as the backend solver.

Table 3 shows the results of checking the machine Movement for timed deadlocks with both FDR4 and CP. We observe that FDR4 is unable to use partial order reduction for this model. Our results show that CP performs better than FDR4 for all values of D considered, indicating that CP can play a role in verification of liveness properties, such as, timed deadlock freedom. On the other hand, CP can only identify violations up to a given horizon.

	FDR4			CP Toolchain	
D	States	Transitions	Time (s)	SR Time (s)	Solv Time (s)
0	2,269,830	5,033,595	169.16	14.92	0.00
10	10,448,306	24,728,591	430.64	54.60	0.00
20	10,761,506	25,464,151	442.91	29.61	0.00
40	11,387,906	26,935,271	460.04	32.24	0.00
80	12,640,706	29,877,511	515.94	32.29	0.00
100	13,267,106	31,348,631	538.25	32.47	0.00
200	16,399,106	38,704,231	662.64	36.16	0.00
400	22,663,106	53,415,431	892.84	41.24	0.00
800	35,191,106	82,837,831	1394.00	56.45	0.00
1000	41,455,106	97,549,031	1634.13	63.87	0.00

Table 3: Results of verifying timed deadlock freedom of the alpha algorithm using FDR4 and CP, varying the parameter D to vary the size of the state space.

8 Related Work

Our work is essentially a form of bounded model checking (BMC). BMC is a key application area for SAT solvers, to verify hardware [4, Ch.18] and software [4, Ch.20]. In BMC, a formula is constructed representing a bounded-length counterexample trace of a given property. The formula is then solved (typically using a SAT solver), either producing a counterexample trace or a proof that no such trace exists (assuming correctness of the encoding). BMC can revisit states, so it can in some cases be less efficient than conventional unbounded model checking.

CP has been applied to verification problems, including bounded program verification. For example, CPBPV [8] (constraint programming for bounded program verification) verifies a given computer program with respect to a formal specification. The framework uses constraint stores which are incrementally updated according to non-deterministic symbolic execution. CPBPV is able to explore execution paths of a bounded length; it avoids spurious execution paths, pruning them early when the constraint store becomes inconsistent. By employing a sequence of solvers in sequence (including MIP and CP solvers), starting with faster and less generalised ones, CPBPV is able to verify programs many times faster than previous bounded-length CP-based approaches. Delzanno and Podelski [9] implicitly represent the set of reachable states as a constraint store in constraint logic programming (CLP), iteratively extending the set by applying the transition function until the least fixpoint is reached. Their approach is not BMC, and does not construct traces. Similarly, CLPS-B [6] uses CLP constraint stores to represent sets of concrete states implicitly. CLPS-B enables animation of B specifications, which is akin to BMC.

Hallerstede and Leuschel [13] extend ProB with a specialised constraint solver for finding deadlock states (and other similar tasks), where a deadlock state satisfies all axioms and invariants, but falsifies all guards. In contrast to our

approach, this form of constraint-based deadlock checking does not construct a trace, so the deadlock state may not be reachable from the initial state.

Krings and Leuschel's symbolic model checking [17] is very closely related to our work. They propose a BMC method for B and Event-B models that uses a constraint solver (in ProB), as well as inductive proof methods related to BMC that are complete (i.e. unbounded). Recently the ProB constraint solver has been integrated with SAT to extend its reach [19]. APALACHE [18,16] is a symbolic model checker (avoiding explicit enumeration of states) for the TLA$^+$ language. It uses the SMT solver Z3 and can generate BMC formulas that have a similar high-level structure to our CP models. Alloy 6 [31] now has mutable state variables. Actions can be expressed in first-order logic, and the Alloy Analyzer can generate BMC formulas in SAT as one of its verification methods.

9 Conclusions

We have explored the formulation of state-based models in CP, encompassing multiple finite-state automata (synchronised on transitions), clocks and timed transitions, non-determinism, and external events acting as triggers. We present a systematic and automatable translation from state-based models to CP. CP also offers a rich language for stating properties to verify. The CP approach could possibly be more general than traditional model-checking because properties can refer to any element of the state at any time step, and make unlimited use of the logic and arithmetic offered by a CP modelling language. The CP models have a time horizon, and so any verification outcomes (such as deadlock freedom) only apply within the time horizon. However, we found that a CP toolchain (i.e. the modelling tool Savile Row with the SAT solver Kissat or CP solver OR-Tools) can scale substantially better than FDR4 in some cases, particularly when partial order reduction does not apply.

As part of our future work, we intend to investigate complete verification of properties with CP. If the horizon is sufficiently large (for example, to ensure that all states of the system can be reached within the horizon), then a given property can be fully verified. Automatically obtaining a suitable value for the horizon is challenging but we can draw on existing work in bounded model checking. Also, we intend to automate the translation to CP, and translation of solutions (traces) back to the original state-based model, in order to avoid errors and improve comprehensibility.

Acknowledgments

RoboChart icons have been made by Sarfraz Shoukat, Freepik, Google, Icomoon and Madebyoliver from `www.flaticon.com`, and are licensed CC 3.0 BY. This work was supported by EPSRC grant EP/W001977/1.

References

1. Akgün, O., Frisch, A.M., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Conjure: Automatic generation of constraint models from problem specifications. Ar-

tificial Intelligence **310** (2022). https://doi.org/10.1016/j.artint.2022.103751
2. Baxter, J., Ribeiro, P., Cavalcanti, A.: Sound reasoning in tock-CSP. Acta Informatica **59**, 125–162 (2022). https://doi.org/10.1007/s00236-020-00394-3
3. Biere, A., Fleury, M.: Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In: Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2022-1, pp. 10–11. University of Helsinki (2022), https://hdl.handle.net/10138/359079
4. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2021)
5. Bjerknes, J.D., Winfield, A.F.T.: On fault tolerance and scalability of swarm robotic systems. In: Martinoli, A., Mondada, F., Correll, N., Mermoud, G., Egerstedt, M., Hsieh, M.A., Parker, L.E., Støy, K. (eds.) Distributed Autonomous Robotic Systems - The 10th International Symposium, DARS 2010, Lausanne, Switzerland, November 1-3, 2010. Springer Tracts in Advanced Robotics, vol. 83, pp. 431–444. Springer (2010). https://doi.org/10.1007/978-3-642-32723-0_31
6. Bouquet, F., Legeard, B., Peureux, F.: CLPS-B - a constraint solver for B. In: Katoen, J.P., Stevens, P. (eds.) Proceedings TACAS'02. LNCS, vol. 2280, pp. 188–204. Springer-Verlag (2002)
7. Cheng, K., Krishnakumar, A.S.: Automatic functional test generation using the extended finite state machine model. In: Dunlop, A.E. (ed.) Proceedings of the 30th Design Automation Conference. Dallas, Texas, USA, June 14-18, 1993. pp. 86–91. ACM Press (1993). https://doi.org/10.1145/157485.164585
8. Collavizza, H., Rueher, M., Van Hentenryck, P.: CPBPV: A constraint-programming framework for bounded program verification. Constraints **15**(2), 238–264 (Apr 2010). https://doi.org/10.1007/s10601-009-9089-9
9. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. STTT **3**(3), 250–270 (2001)
10. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (jun 2008). https://doi.org/10.1007/s10601-008-9047-y
11. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.: FDR3 — A Modern Refinement Checker for CSP. In: Ábrahám, E., Havelund, K. (eds.) Proceedings TACAS'14. LNCS, vol. 8413, pp. 187–201 (2014)
12. Gibson-Robinson, T., Hansen, H., Roscoe, A.W., Wang, X.: Practical partial order reduction for CSP. In: NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9058, pp. 188–203. Springer (2015). https://doi.org/10.1007/978-3-319-17524-9_14
13. Hallerstede, S., Leuschel, M.: Constraint-based deadlock checking of high-level specifications. Theory Pract. Log. Program. **11**(4–5), 767–782 (2011)
14. Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987). https://doi.org/10.1016/0167-6423(87)90035-9
15. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Software Eng. **23**(5), 279–295 (1997). https://doi.org/10.1109/32.588521
16. Konnov, I., Kukovec, J., Tran, T.: TLA+ model checking made symbolic. Proc. ACM Program. Lang. **3**(OOPSLA), 123:1–123:30 (2019). https://doi.org/10.1145/3360549

17. Krings, S., Leuschel, M.: Proof assisted bounded and unbounded symbolic model checking of software and system models. Sci. Comput. Program. **158**, 41–63 (2018). https://doi.org/10.1016/j.scico.2017.08.013
18. Kukovec, J., Tran, T., Konnov, I.: Extracting symbolic transitions from TLA$^+$ specifications. In: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th International Conference, ABZ 2018, Southampton, UK, June 5-8, 2018, Proceedings. pp. 89–104 (2018). https://doi.org/10.1007/978-3-319-91271-4_7
19. Leuschel, M.: B2SAT: a bare-metal reduction of B to SAT. In: International Symposium on Formal Methods. pp. 122–139. Springer (2024)
20. McEwan, A., Woodcock, J.: A refinement based approach to calculating a fault-tolerant railway signal device. In: Proceedings of the 18th IFIP World Computer Congress, Topical Day on Verified Software. Springer (2004)
21. Mealy, G.H.: A method for synthesizing sequential circuits. The Bell System Technical Journal **34**(5), 1045–1079 (1955). https://doi.org/10.1002/j.1538-7305.1955.tb03788.x
22. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J., Woodcock, J.: RoboChart: Modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling **18**, 1–53 (10 2019). https://doi.org/10.1007/s10270-018-00710-z
23. Moore, E.F., et al.: Gedanken-experiments on sequential machines. Automata studies **34**, 129–153 (1956)
24. Murray, T.C.: On the limits of refinement-testing for model-checking CSP. Formal Aspects Comput. **25**(2), 219–256 (2013). https://doi.org/10.1007/S00165-011-0183-6
25. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Proceedings of 13th International Conference on Principles and Practice of Constraint Programming CP-2007. pp. 529–543 (2007)
26. Nightingale, P., Özgür Akgün, Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
27. Perron, L., Didier, F.: OR-Tools CP-SAT, version 9.11 (2024), https://developers.google.com/optimization/cp/cp_solver/
28. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Intl. Conf. on principles and practice of constraint programming. pp. 751–755. Springer (2006)
29. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
30. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
31. Sinha, S., Kang, E.: Formal modeling and analysis of apache kafka in alloy 6. In: Proceedings of ABZ 2024. pp. 25–42 (2024). https://doi.org/10.1007/978-3-031-63790-2_2
32. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA (1999)
33. Woodcock, J.C.P., Davies, J.: Using Z - specification, refinement, and proof. Prentice-Hall (1996)
34. Yan, F., Foster, S., Habli, I.: Automated compositional verification for robotic state machines using Isabelle/HOL. In: Aït-Ameur, Y., Khendek, F., Méry, D. (eds.) 27th International Conference on Engineering of Complex Computer Systems, ICECCS. pp. 167–176. IEEE (2023). https://doi.org/10.1109/ICECCS59891.2023.00029