
Proceedings Paper:
YAN, FANG, Ballenghien, Benoît, FOSTER, SIMON DAVID et al. (2026) Automated Verification of Robot Software Models with Assume-Guarantee Reasoning in Isabelle/HOL. In: 17th International Conference on Interactive Theorem Proving (ITP 2026). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany.

https://doi.org/10.4230/LIPIcs.ITP.2026.10

Automated Verification of Robot Software Models with Assume-Guarantee Reasoning in Isabelle/HOL

Fang Yan
University of York, UK

Benoît Ballenghien
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, France

Simon Foster
University of York, UK

Ana Cavalcanti
University of York, UK

James Baxter
University of York, UK

Burkhart Wolff
Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, France

Abstract

We present a theorem-proving-based technique for verifying deadlock freedom of CSP-style concurrent models in Isabelle/HOL. The approach addresses challenges that are difficult to handle using model checking alone, including infinite state spaces, compositional reasoning in the presence of shared variables, and the need for mechanised proofs. Our main contribution is a coinductive characterisation of deadlock freedom that is equivalent to the standard CSP refinement-based definition, but is more amenable to automated reasoning in an interactive theorem prover. To support reasoning about shared variables, we introduce an assume–guarantee strategy that enforces invariants within transition semantics. The technique is generally applicable to CSP specifications that model shared variables using standard CSP constructs. In particular, we consider the semantics of RoboChart, a domain-specific modelling language for robotic control software, which we mechanise in Isabelle via a shallow embedding in HOL-CSP, and implement automated proof methods. The approach is evaluated on three case studies, including two RoboChart models of industrial robotic systems.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases process algebra, Isabelle/HOL, automated proof methods, deadlock freedom

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.10

Supplementary Material
Software (Formalisation): `https://github.com/laila-fangyan/RoboChart-Deadlock-HOLCSP-AG`
archived at `swh:1:dir:339c295e6478e8704c28da8404687ca55587ebd2`

Funding The work is partially funded by the UK EPSRC Grants EP/R025479/1 and EP/V026801/2, by the Royal Academy of Engineering Grants No CiET1719/45 and IF2122\183 and by the RoboSAPIENS project funded by the European Commission's Horizon Europe programme under grant agreement number 101133807.

Declaration about GenAI/LLM use Generative AI tools were used only for language polishing. All technical content, proofs, formalisation, and scientific contributions were produced by the authors.

1 Introduction

Theorem proving offers greater scalability and flexibility than model checking, enabling the verification of systems with complex structures and infinite state spaces. For robotic systems, where controllers often feature rich state machines and unbounded data types, and may involve concurrent behaviours, these capabilities are key. In this paper, we present an automated approach to prove deadlock freedom of Communicating Sequential Processes (CSP) models [31] using the Isabelle/HOL theorem prover [27]. Automation is enhanced by considering patterns of modelling for parallel state machines with shared variables. The pattern is highly relevant for modelling controllers, and, in particular, it is used in the semantics of RoboChart [24], a domain-specific modelling language for robot controllers. Its formal semantics is defined using the state-rich process algebra *Circus* [29], which combines CSP and Z [37]. We focus on proof of deadlock freedom, but our results are relevant as a foundation for a more general invariant-based verification technique.

Our work addresses three main challenges, including reasoning about infinite state spaces, supporting compositional reasoning in the presence of shared variables, and enabling automation. *Infinite state spaces* arise from unbounded data domains or recursive behaviours. Systems exhibiting them are often beyond the reach of traditional model checking due to the state-space explosion problem. We can overcome this with the symbolic fixed-point definition of deadlock freedom [31], but this is not amenable to automated proof. In this paper, we therefore mechanise a coinductive specification of deadlock freedom that avoids constructing an explicit transition system for models, and provides a structure for automated proofs. This coinductive verification approach enables verification to scale naturally to infinite-state systems such as RoboChart controllers. We show that a process refines our coinductive specification, if, and only if, it is deadlock-free according to the fixed-point definition.

Previous theorem-proving works for RoboChart [13, 38] do not address compositional reasoning in the presence of shared variables. We address this challenge with a specialised assume–guarantee reasoning framework. We assume that the environment (robot and other controllers) preserves a given invariant on shared variables, ensuring that some event remains enabled in all reachable configurations. Symmetrically, each component under verification must guarantee that its updates preserve the invariant. With this, deadlock freedom can be established even in the presence of *shared variables*. The assume–guarantee structure also provides a foundation for extending the approach to compositional reasoning about concurrent systems. In this work,we focus on the verification of individual state-machine-based models.

The third challenge is the automation of RoboChart verification. Whilst there have been efforts to mechanise *Circus* in theorem provers [12, 14, 36], support for automated reasoning is limited. For that, we exploit the HOL-CSP framework [4, 5], a library mechanising denotational definitions of CSP operators together with formally derived algebraic and refinement laws. The *Circus* semantics of RoboChart can be encoded in CSP, as has been done to enable model checking with FDR4, the Failures-Divergences Refinement checker [16]. So, with HOL-CSP we leverage Isabelle's proof facilities to enable the automated verification of the properties of the resulting CSP processes, given suitable user-provided invariants. This approach allows us to perform rigorous formal analysis while retaining the intuitive engineering abstractions provided by RoboChart's modelling language.

Furthermore, we demonstrate the practical utility of our approach through four diverse examples. Unlike previous RoboChart verification efforts [13, 14, 38] that were limited to small-scale examples, we evaluate our coinductive characterisation across a broader range of robotic patterns. Besides a running example, we consider an autonomous chemical detector,

an underwater vehicle, and a TurtleBot navigation system. By addressing these non-trivial cases, we show that our technique scales to handle essential verification challenges, such as complex state-space interactions, which were not fully explored in prior work.

This paper makes the following contributions. (1) We define a coinductive, transition-level characterisation of deadlock freedom for CSP-style state machines and formally prove its equivalence to the standard CSP refinement-based definition of deadlock-freedom in Isabelle/HOL. (2) Based on this characterisation, we develop an automated proof method that discharges deadlock-freedom proof goals by normalisation and refinement-based reduction. (3) We implement a systematic shallow embedding of RoboChart transition semantics into HOL-CSP, enabling push-button verification of deadlock freedom for state-machine models with shared variables. (4) We apply our approach to three case studies and one running example, demonstrating the applicability and effectiveness of our approach.

Next, § 2 provides a description of RoboChart and Isabelle HOL-CSP. § 3 gives our coinductive definition of deadlock freedom. § 4 presents our approach to assume-guarantee reasoning. § 5 provides a novel set of deductive proof rules for the CSP operators, mechanised in HOL-CSP. § 6 introduces the automated proof procedure. § 7 presents case studies. § 8 discusses related work, and § 9 summarises and describes future work.

2 Background

In this section, we present RoboChart and Isabelle HOL-CSP.

RoboChart can be seen as a profile of UML state machines with extensions for real-time modelling and concurrency. It supports a component structure in which a *robotic platform* declares services of the robot needed by the software, and a set of (parallel) *controllers* represents the (concurrent) software. In this paper, we focus on the behaviour of controllers, which are specified by state machines that may interact via shared variables. The semantics structure of a RoboChart state machine is illustrated in Fig. 1.

A RoboChart state machine describes system behaviour in terms of states and transitions between them. Intuitively, it captures how the system moves between states in response to events. Formally, it is defined as a directed graph comprising nodes and transitions. The nodes can be initial junctions, simple or composite states, junctions, and final states, which define the termination of the state machine. Transitions define the control flow from the source nodes to the target nodes. A transition is initiated by a *trigger* (an event), but an optional *Boolean guard* (a logical condition) can control whether it is enabled. Crucially, a transition can only be taken if the trigger occurs at a moment when the guard evaluates to true. If the guard is false, the transition remains disabled. When taken, its associated actions are executed, if they exist, before entering the target node.

The semantics for a state machine is given as the parallel composition of processes for its nodes and for its transitions. While the semantics of nodes is given by a parallel composition, the execution follows a sequential control flow, with exactly one simple active (leaf) node. Composite states, which may have multiple active nodes are not addressed here.

RoboChart state machines must declare the variables that it uses; for local variables, an initial value may be given upon declaration; if not, the *Circus* semantics defines a default value, recorded in the corresponding memory process. These memory processes for a state machine hold both the local and required (shared) variables. Transition actions may update shared variables, and guards may depend on their values. These features make transitions the critical points where concurrency, communication, and data-dependent behaviour converge, and thus central to the verification of properties such as deadlock freedom. For a comprehensive account of the syntax and semantics of RoboChart, we refer to [25].

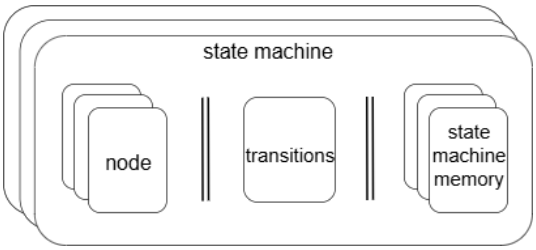

Figure 1 An illustration of semantics of a RoboChart state machine adapted from [24, Fig. 11] and [9, Fig. 49]. Parallel composition is represented by stacked components or by parallel lines.

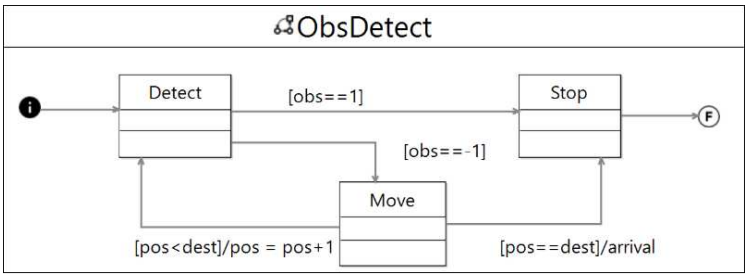

Figure 2 RoboChart of the ObsDetect state machine.

HOL-CSP is a mechanisation of CSP in Isabelle/HOL, covering denotational, algebraic, and (more recently) operational facets of the CSP semantics. The full range of CSP operators commonly found in the literature is supported, including prefixing, external and internal choice, parallel composition, hiding, interruption, and recursion. We describe them as needed. At the denotational semantic level, the failures/divergences model is implemented directly. Then, the traces, failures, and divergences can be recovered through natural projections. Algebraic and operational laws are formally proved, together with refinement properties.

Running example. To illustrate our approach, we use an obstacle-avoiding robot that moves towards a destination. Its behaviour is modelled in RoboChart. The context declares a shared variable obs, with values 1 and -1 indicating the presence and absence of an obstacle, respectively, and a constant dest for the destination. The state machine ObsDetect, as shown in Fig. 2, models the robot's behaviour, using a local variable pos to record its position, incremented by one after each move. The machine has five nodes. The initial junction (black circle with a i), indicates that the control flow starts at the Detect state. If obs is 1, the machine transitions to the state Stop; otherwise, it goes to the state Move. In Move, if we have that pos == dest, the event arrival is raised; otherwise, pos is increased by 1.

3 Coinductive deadlock freedom

We focus on the verification of RoboChart state machines to establish deadlock freedom, ensuring that the system is always capable of progress unless it has reached a designated final state. In RoboChart, a deadlock occurs at a *non-final* state of a machine if it has no outgoing transitions, or if all the transitions are disabled because their guards evaluate to false. Also, a transition action deadlocks if it applies a function outside of its precondition. Since guards may depend on both local and shared variables, we refer to them collectively as the machine's *state variables*. Assumptions about these variables can be captured as invariants. For our running example, for instance, obs can take only values in $\{-1, 1\}$.

We introduce below the notion of a *state context*, which represents a valuation of all state variables and provides the basis for defining invariants and guards.

▶ **Definition 1.** *Let $x_1, \ldots, x_n$ be the state variables of a state machine. A* state context *is a valuation $\Gamma = \{x_1 = v_1, \ldots, x_n = v_n\}$.*

Based on this, we can now define coinductive deadlock freedom for transitions.

▶ **Definition 2.** *A state machine's transition behaviour is* deadlock-free under an invariant I *if, and only if, for every non-terminal node (that is, a node that is not a final state) n and for every state context Γ satisfying I, written $\Gamma \vDash I$, there exists at least one enabled outgoing transition from n such that the action of that transition executes successfully and the resulting state context Γ' also satisfies I ($\Gamma' \vDash I$).*

Here, the invariant I is a predicate over Γ, determined by the intended ranges of variables as illustrated above. If the guards of the outgoing transitions of a node jointly cover all valuations, then $I \equiv \mathsf{True}$; otherwise, I records admissible ranges assumed by the model. Thus, verification of deadlock freedom is carried out *under* the assumption I.

We consider a non-terminal node n with outgoing transitions indexed by $i \in \{1, \ldots, k\}$, where e_i are the trigger events, g_i are the guards, and A_i are the associated actions. In our definition, the trigger events e_i are regarded as available: their enabledness does not affect the outcome of the deadlock-freedom check. Event availability needs to be taken into account only when reasoning at the state-machine or higher level, where event communication is involved. So, the deadlock freedom of transitions under I requires that at least one g_i holds in the current context Γ and that A_i maps Γ to a context that still satisfies I.

By requiring this condition to hold for every non-terminal node, the entire transition system of the state machine is deadlock free. This is a *coinductive* characterisation of deadlock freedom: it assumes I in the current context and requires each taken transition to preserve I, thereby establishing deadlock freedom across all reachable configurations.

The value of Γ, recorded in memory processes, determines the truth of g_i. As we explain in § 4, the transition processes include assumptions on Γ that determine the values of the guards. So the coinductive definition requires only that I is preserved by the transitions and assumes the environment satisfies the assumption. Whether I holds initially and is preserved by the environment are obligations to be discharged for the environment.

For our running example, I is $\mathsf{obs} \in \{-1, 1\} \wedge \mathsf{pos} \leq \mathsf{dest}$, and deadlock freedom can be argued as follows. (1) *Initial i:* there is a single unconditional transition to Detect. (i) The guard is always true, so it is enabled whenever $\Gamma \vDash I$. (ii) This transition has *no action*, hence $\Gamma' = \Gamma$ and I is preserved. (2) *Stop:* similar. (3) *Detect:* there are two outgoing transitions with guards $\mathsf{obs} = 1$ and $\mathsf{obs} = -1$. (i) Under I (which requires $\mathsf{obs} \in \{-1, 1\}$), exactly one guard holds, so at least one transition is enabled. (ii) Both transitions have *no action*, hence $\Gamma' = \Gamma$. (4) *Move:* two cases arise. (i) If $\mathsf{pos} == \mathsf{dest}$, the action of the transition is an event communication that does not change the state, so I is trivially preserved. (ii) If $\mathsf{pos} < \mathsf{dest}$, the transition enabled increments pos by 1. Since I requires $\mathsf{pos} \leq \mathsf{dest}$ and $\mathsf{pos} < \mathsf{dest}$ here, we have $\mathsf{pos} + 1 \leq \mathsf{dest}$, so I is preserved; obs remains in $\{-1, 1\}$ by assumption. Therefore, every non-final state has at least one enabled transition whose (possibly empty) action preserves I; hence, by the definition above, the transition behaviour is deadlock-free under I.

Our verification approach attempts to prove that RoboChart models satisfy coinductive deadlock freedom and invariants, using Isabelle. We have implemented a model transformation from RoboChart to HOL-CSP processes, preserving the *Circus* semantics like it is done for the semantics for model checking with FDR. Next, we describe how the generated HOL-CSP semantics is automatically enriched into a format convenient for proof.

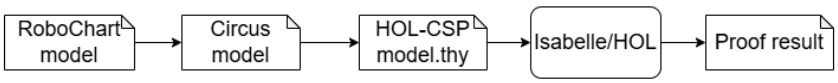

Figure 3 Overview of the verification pipeline from RoboChart models to Isabelle/HOL proofs.

4 Semantics enrichment

HOL-CSP generation. To obtain a HOL-CSP specification that can be used to verify a RoboChart model in Isabelle, we have developed a general model-transformation algorithm that comprises two-stages. The first stage is a systematic mapping from RoboChart metamodel elements to *Circus* metamodel elements. The mapping is faithful to the semantics functions defined for RoboChart in [25]. The second stage is a translation from *Circus* constructs to HOL-CSP notations. For example, RoboChart states are mapped to *Circus* processes, and transitions are encoded as a process containing prefixed external choices. The data structures, including types and channels, are automatically obtained from the RoboChart models. Then, these are further translated into HOL-CSP notations. We have implemented this general model-transformation algorithm in Eclipse using the Epsilon Object Language (EOL) [21] for the first model-to-model stage and the Epsilon Generation Language (EGL) [32] for the second model-to-text stage, so that the RoboChart models can be automatically translated into an Isabelle/HOL theory, as illustrated in Fig. 3.

The RoboChart semantics is compositional, so that our model-transformation algorithm is applicable to RoboChart components as well as complete models. As already mentioned, in this work, we focus on the HOL-CSP process that captures the transition behaviour of RoboChart state machines (see Fig. 1). By applying the transformation method discussed above, the transitions of the state machine ObsDetect (depicted in Fig. 2) are automatically mapped to the HOL-CSP process `Trans`, as shown in Listing 1.

The `Trans` process in Listing 1 is inside an Isabelle *locale*, a construct used as a container for the definitions. `Trans` begins with an input of the shared variables `obs` and `pos` (line 6) via `get` channels, and then proceeds as an external choice (□) between several guarded branches. Each branch corresponds to one transition, with a guard such as `obs = -1` or `pos < dest` determining when the branch is enabled. For instance, in line 11, the branch guarded by `pos < dest` models the behaviour in state Move when the robot has not yet reached its destination: it increments `pos` by one and recurses to `Trans`. The remaining branches follow the same pattern, each capturing one transition of ObsDetect.

Each branch includes communications on events that encode the RoboChart operational semantics: `internal_.NID_*` mark control flow points, for example, `internal_.NID_move` (line 11) indicates that the model is at the state `Move`, which is the source node of the transition captured by the branch of the external choice in line 11; and `enter_*`, `exit`, and `exited` capture state entry and exit, for example, `enter_ Detect` (line 11) indicates the transition leads to the state Detect. The process `SSTOP` (line 5), defined as a synchronisation on an event `share`, is combined with other processes through the interrupt operator (△) to implement the CSP protocol for shared-variable access. Finally, the recursive call to `Trans` (line 12) ensures that the state machine continues to react to further inputs, controlled by transitions. This is a direct encoding of the CSP semantics of RoboChart [25].

Semantics enrichment. To support (deadlock-freedom) proof, we enrich the *Circus* semantics of RoboChart in three ways: parametrisation, final state handling, and assume–guarantee reasoning. These enrichments do not change the original *Circus* semantics, but make the

Listing 1 HOL-CSP model of Transitions behaviour of ObsDetect state machine in Fig. 2.

```
locale <Trans> begin
fixrec
SSTOP  :: "chan_event process" and
Trans :: "chan_event process"  where
"SSTOP = share  →  SSTOP" |
"Trans = (SSTOP △ (get_obs?obs → (get_pos?pos → (
    ( ((internal_.NID_i0 → Skip); (enter_detect → Skip))
    □ ...
    □ ((internal_.NID_stop → Skip); (SSTOP △ (exit → Skip)); SSTOP △ ((exited → Skip);
        (enter_f0 → Skip)))
    □ (pos = dest) & ((internal_.NID_move → Skip); (SSTOP △ (exit → Skip)); (SSTOP △
        ((exited → (SSTOP △ (arrival_out → Skip))); (enter_stop → Skip))))
    □ (pos < dest) & ((internal_.NID_move → Skip); (SSTOP △ (exit → Skip)); (SSTOP △
        ((exited → (SSTOP △ (get_pos?pos → (SSTOP △ (set_pos!(pos + 1) →
        Skip)))));(enter_detect → Skip))))))
    ; Trans))))"
end
```

Listing 2 Enriched Trans. (purple: assume–guarantee; blue: node parametrisation / final.)

```
locale <Trans> begin
abbreviation "assume b Q P ≡ (if b then P else aviol → Q)"
abbreviation "guar b P ≡ (if b then P else gviol → STOP)"
fixrec
SSTOP  :: "NIDS_ObsDetect ⇒ chan_event process" and
Trans :: "NIDS_ObsDetect ⇒ chan_event process"  where
"SSTOP = share  →  SSTOP" |
"Trans·n =
  (SSTOP △ (get_obs?obs → (get_pos?pos →
   (assume ((obs = 1 ∨ obs = -1) ∧ pos ≤ dest) Trans·n (
      (n = NID_i0) & ((internal_.NID_i0 → Skip); (enter_detect → Trans·NID_detect))
    □ ...
    □ (n = NID_stop) & ((internal_.NID_stop → Skip); (SSTOP △ (exit → Skip)); SSTOP △
        ((exited → Skip); (enter_f0 → Trans·NID_f0)))
    □ (n = NID_move)& (pos = dest) & ((internal_.NID_move → Skip); (SSTOP △ (exit →
        Skip)); (SSTOP △ ((exited → (SSTOP △ (arrival_out → Skip))); (enter_stop →
         Trans·NID_stop))))
    □ (n = NID_move)& (pos < dest) & ((internal_.NID_move → Skip); (SSTOP △ (exit →
        Skip)); (SSTOP △ (exited → (SSTOP △ (get_pos?pos → (assume ((obs = 1 ∨ obs =
        -1) ∧ pos ≤ dest) Trans·n (SSTOP △(guar ((obs = 1 ∨ obs = -1) ∧ pos ≤ dest)
       (set_pos!(pos+1) → enter_detect → Trans·NID_detect))))))))))
    □ (n = NID_f0) & (terminate → Skip)))))"
end
```

resulting HOL-CSP models more amenable to mechanised verification. The enriched `Trans` of the running example is given in Listing 2. Next, we describe how `Trans` is changed by each enrichment.

Enrichment 1: Parametrisation of the active node. In the original semantics, the active node is not represented as a variable but tracked implicitly through synchronisation on `internal_` and `enter_`, `exit`, and `exited` events. For example, in Listing 1, `internal_.NID_move` indicates that the transition with the guard `pos < dest` originates from Move. So, transitions are guarded by their source node, but there is no state variable to record the active node.

We therefore enrich the semantics by introducing the active node as an explicit parameter `n` of the recursive action `Trans`, and by adding guards of the form `n = src` to each branch of the external choice. For example, in Listing 2 line 15, the transition modelled corresponds to that in Listing 1 line 11, but now with an explicit guard `n = NID_move`. This transformation does not alter the behaviour of `Trans`, which already enforces the same restriction. Its benefit is that structured reasoning becomes possible: proofs can proceed by case analysis on the parameter `n`, directly reflecting the structure of the state machine.

Enrichment 2: Final-state handling. In the original *Circus* semantics, the behaviour after entering a final state is specified by the semantics of the final state itself; it is not captured in `Trans`. For example, in Listing 1 line 9, the event `enter_f0` denotes an entry into the final state `f0`, but the subsequent termination is defined elsewhere. So, when we parametrise `Trans` by the active node and reason by case analysis on `n`, there is no branch for `n = f0`, which makes the case split incomplete. We therefore enrich the semantics with an explicit transition branch for final states in `Trans` (see Listing 2, line 17). This preserves the semantics of termination while integrating it into the same parametric structure as other transitions, enabling uniform case analysis and smooth coinductive proofs.

Enrichment 3: Assume–guarantee reasoning. We enrich the *Circus* semantics of shared variable access with explicit assume and guarantee checks. For that, two new events are introduced: `aviol` signals an assumption violation, and `gviol` signals a guarantee violation. Two processes use these events (see Listing 2, lines 2–3). The behaviour of the process `assume b Q P` is as follows. If the condition `b` holds, `assume b Q P` behaves as `P`; otherwise, `assume b Q P` raises `aviol` and continues with the fallback process `Q`. As for `guar b P`, if `b` holds, `guar b P` proceeds as `P`; otherwise, `gviol` is raised and `guar b P` deadlocks (`STOP`). We use `assume` to annotate variable accesses, and `guar` to annotate updates.

Besides the variable inputs that may occur within the body of a transition, `Trans` also specifies that the external choice over transition branches is preceded by an input of all variables (see Listing 2, line 9). Accordingly, the first `assume` is inserted immediately after this initial input to record the assumption that the input values satisfy the invariant. Its continuation `P` corresponds to the remaining behaviour from line 11 to line 17 in Listing 2. For the fallback, we always use simply `Trans(n)` itself, so that `Trans` recurses and accepts the next input if the assumption is violated. Such violations of assumptions are identified as caused by the environment, not by the component under consideration.

The continuation after an input is always the process `P` that defines the behaviour of the model following the variable input up to the point where the associated transition is completed. Identifying this process requires some behaviour preserving simplification of the structure of `Trans`. For example, a variable input occurs in the original model (Listing 1, line 11). This input is embedded within the following CSP process (in the external choice).

```
(exited→(SSTOP△(get_pos?pos→(SSTOP△...Skip))));(enter_detect→Skip)
```

To introduce an appropriate point to insert an `assume` process for the input of `pos` via the channel `get_pos`, we rely on laws of CSP to distribute the sequential composition into the preceding process. After that, the branch becomes as follows.

```
exited→(SSTOP△(get_pos?pos→SSTOP△...→enter_detect →Skip))
```

Once the `assume` process is inserted after the input `get_pos?pos`, the continuation is

```
SSTOP△... →enter_detect→Trans·NID_detect
```

For recording the guarantees, whenever there is a variable update (via a `set_` event), we insert a `guar` process. For continuation, we use the process that starts from the `set_` communication to the end of the updating scope, that is, the process prefixed by the `set_` communication. As said, in this case, there is no fallback; if there is a violation, it leads to `STOP`. For example, line 15 of Listing 2 illustrates the `guar` annotation for the `pos` update, which enforces the specified invariant throughout the operation.

The enrichments that introduce the `assume` and `guar` processes are conservative: when all assumptions and guarantees are satisfied, the behaviour coincides exactly with that of the original process. The role of the annotations is to expose contract violations as explicit events, making them observable during verification. In particular, component-level proofs of deadlock freedom are carried out only over traces that satisfy all assumptions. Guarantee violations, by contrast, represent internal errors and are treated as genuine deadlocks.

This assume-guarantee mechanism corresponds to the classical Rely-Guarantee (RG) reasoning framework [19]. In our setting, the `assume` condition $I(\vec{v})$ is a *Rely-condition*, encoding the expectations placed on the environment (the shared variables), while the `guar` checks act as *Guarantee-conditions* that the component must satisfy. By establishing this contract-style boundary, we achieve a compositional verification strategy: we can reason about deadlock freedom of an individual controller by assuming the environment respects I, bypassing the need to construct the combined state space of the entire system.

Following the enrichments above, the `Trans` process has the following prototypical form.

▶ **Definition 3** (`Trans` Prototypical Form)**.**

$$\begin{aligned}&\texttt{Trans(n)} = \texttt{SSTOP}\triangle \texttt{get}_{\underset{x}{\rightarrow}}?\vec{v} \rightarrow \texttt{assume}\ (I(\vec{v}))\ (\texttt{Trans(n)})\\ &\quad(\underset{t\in\mathcal{T}}{\Box}\ \texttt{n} = t.src\ \&\ t.guard\ \&\ t.trigger \rightarrow t.action; \texttt{enter\_}t.tgt \rightarrow \texttt{Trans}(t.tgt))\end{aligned}$$

Here, `n` is the explicit parameter recording the active node. $\texttt{SSTOP}\triangle\ \texttt{get}_{\underset{x}{\rightarrow}}?\vec{v}$ captures part of a shared-variable protocol whose details are not relevant here, except to note that $\texttt{get}_{\underset{x}{\rightarrow}}?\vec{v}$ inputs the current values $\vec{v}$ of all state variables $\vec{x}$. This is followed by an `assume` process, whose condition $I(\vec{v})$ is the invariant over these variables. This invariant is already embedded in the enriched semantics; proof rules (such as Asm-Red) later make this assumption explicit in the proof context. The replicated external choice $\Box_{t\in\mathcal{T}}$ ranges over the set $\mathcal{T}$ of transitions. For each transition t, the guard $\texttt{n} = t.src$ ensures that just the transitions from the active node are considered, $t.guard$ is the transition condition (omitted if it is `True`), and $t.trigger$ is the trigger event (if it exists), or `internal_.NID(n)`, if it does not. The term $t.action$ represents the sequence of observable events implementing the transition action, including any nested `assume` and `guar` checks. Finally, `enter.`$(t.tgt)$ records entry into the target node, and the recursive call `Trans(`$t.tgt$`)` restarts the transition with the new active node.

In summary, `Trans` first retrieves all state variables and asserts that the invariant I holds, and then behaves as a replicated external choice over all enabled transitions from the current node. Deadlock-freedom proofs are therefore carried out *under the assumption* that I holds. System-level deadlock freedom requires checking that all component assumptions are mutually consistent. The implementation of the enrichment of the semantics is integrated into the transformation process to generate enriched *Circus* models; subsequently, a standard model-to-text transformation produces the final HOL-CSP model.

5 Deadlock-freedom verification

In this section, we give definitions and theorems needed for our verification strategy. For clarity, we use here the mathematical notation of CSP, explained as needed. Our definitions and theorems are, however, encoded and proved in Isabelle, available in our repository.

We recall the standard definition of deadlock freedom in CSP, where $P \sqsubseteq Q$ is failures-divergences refinement ("P is refined by Q", sometimes denoted $P \sqsubseteq_{FD} Q$), $\sqcap$ is internal choice, and $\Box$ is external choice.

▶ **Definition 4.** *For a process P and a set Σ of events in scope, we define*

$$\mathit{deadlock\_free}\ P \equiv \mathit{DF}\ \Sigma \sqsubseteq P \quad \mathit{where} \quad \mathit{DF}\ A \equiv \mu X.\ \left(\sqcap_{a \in A}\ a \to X\right) \sqcap \mathit{Skip}$$

Here, deadlock freedom is specified as a refinement ($\sqsubseteq$) property, where the specification $DF\ \Sigma$ is a process that can recursively (μ) choose nondeterministically ($\sqcap$) to perform an event from Σ, the set of all events in scope, or terminate successfully (Skip). Above, DF is defined generically for any set of acceptable events A. Note that $deadlock\_free$ is a property of the process itself and does not depend on the alphabet. Also note that the concept of deadlock freedom in CSP (and HOL-CSP) usually does not allow for successful termination; in fact, we are introducing the $deadlock\_free_{SKIPS}$ variant here. However, since only the latter concept is discussed in this paper, we will continue to use this convention.

Proof of deadlock-freedom for recursive CSP processes, such as `Trans`, requires that we formalise a coinduction principle that generalises the principle for RoboChart in Definition 2. We prove that our coinduction principle is sufficient to establish the standard CSP characterisation above. The coinductive specification for deadlock freedom is given below.

▶ **Definition 5** (Coinductive Deadlock-Freedom Specification)**.**

$$\mathrm{DFI}^{\checkmark}_{A}(P) \ \equiv \ \sqcap_{i>0} \left(\sqcap_{\vec{\alpha} \in A^i} \vec{\alpha} \to (P \sqcap \mathit{Skip}) \right)$$

Here, $i > 0$ ensures that all non-empty traces are considered, rather than just single-step executions. A^i is the set of all traces of length i with events in A. So, $\vec{\alpha}$ is such a trace, and $\vec{\alpha} \to P$ is the process that performs the events in $\vec{\alpha}$ in order and then behaves as P.

$\mathrm{DFI}^{\checkmark}_{A}$ is a coinductive specification for deadlock freedom, which closely mirrors the definition of $DF\ A$ in Definition 4. Here, we consider a given process P, and define a process that can nondeterministically choose a size i for a non-empty trace, and then nondeterministically choose to either perform a trace $\vec{\alpha}$ of i events from a given set A before behaving as P, or terminate (Skip). We also define $\mathrm{DFI}^{*\checkmark}_{A}(P)$, which is similar to $\mathrm{DFI}^{\checkmark}_{A}(P)$ except that it allows an empty sequence of events ($i \geq 0$ instead of $i > 0$), and is typically introduced when at least one event has already been registered in a deadlock proof.

$\mathrm{DFI}^{\checkmark}_{A}$ allows us to define a coinduction principle, based on the idea that $\mathrm{DFI}^{\checkmark}_{\Sigma}(P) \sqsubseteq P$ implies that P is deadlock-free. $\mathrm{DFI}^{\checkmark}_{\Sigma}(P) \sqsubseteq P$ requires that there is a non-empty sequence of events that P can perform, before it then behaves as P itself, or terminates. This ensures that P never reaches a state in which no event is enabled, unless it terminates.

We consider, as an example, an event a and the process defined as $As = a \to As$. Intuitively, As can always perform a and then return to the same behaviour, so it is deadlock-free. To use our coinduction principle, we must show $\mathrm{DFI}^{\checkmark}_{\Sigma}(As) \sqsubseteq As$. By definition, $\mathrm{DFI}^{\checkmark}_{\Sigma}(As)$ specifies all behaviours that consist of a non-empty trace of a events followed by As. Since As can perform any finite sequence of a's and then behave again as As, every behaviour allowed by $\mathrm{DFI}^{\checkmark}_{\Sigma}(As)$ is also a behaviour of As. So, the refinement holds.

The next theorem establishes our coinduction principle for parametric systems of recursive processes, such as `Trans.n` in Listing 2. We consider a parametrised process $P(x)$, defined by an equation $P(x) = Q(x)$, where $Q(x)$ may itself be defined in terms of P as in our example. In this case, to establish deadlock freedom, we need to require refinement for all values of the parameter x. So, we apply $\mathrm{DFI}^{\checkmark}_{A}$ to the process that captures all possible behaviours of P, by allowing for an arbitrary (nondeterministic) choice of any argument y. We then require refinement of that process by the body $Q(x)$ of P's definition, for all values of x. In this case, we establish deadlock freedom for any x.

▶ **Theorem 6** (Deadlock-freedom Coinduction). *Let P be a parametric process with $P(x) = Q(x)$ for all x, and V the set of all values, then:*

$$\frac{\Gamma \vdash \forall x.\ \mathrm{DFI}^{\checkmark}_{\Sigma}\left(\sqcap_{y \in V} P(y)\right) \sqsubseteq Q(x)}{\Gamma \vdash \mathit{deadlock\_free}\left(\sqcap_{x \in V} P(x)\right)}$$

The next theorem relates the result in Theorem 6 to Definition 2, providing a justification for the use of our Isabelle encoding in the context of RoboChart.

▶ **Theorem 7.** $\mathrm{DFI}^{\checkmark}_{\Sigma}(P) \sqsubseteq P$ *formalises the transition-level formulation of deadlock freedom in Definition 2, generalising it to CSP processes P, provided P is annotated with invariants as described in the previous section.*

Proof. Here we provide a proof sketch instead of a fully formal proof. Definition 2 states that, in a state machine, every non-terminal node must have at least one enabled outgoing transition whose execution preserves an invariant I. As already illustrated, in the CSP semantics of RoboChart, the execution of a transition is given semantics as a (non-empty) sequence of observable events, capturing the exit from a source state, the transition trigger and the associated transition actions, if any, and the entry in the target state. $\mathrm{DFI}^{\checkmark}_{\Sigma}(P)$ captures exactly this idea: it specifies all processes that can perform at least one event from an alphabet Σ and then continue as P, optionally allowing immediate termination. Thus, as explained, showing $\mathrm{DFI}^{\checkmark}_{\Sigma}(P) \sqsubseteq P$ amounts to proving that whenever a (non-empty) sequence of enabled events exists, P is capable of performing it and reaching a state where it behaves as P. This captures the requirement that transitions always exist whose actions do not deadlock. That all this is established under the invariant I follows from the fact that it has been incorporated into the assumptions of the process P. In particular, if the invariant is broken, the process deadlocks, and so does not satisfy the specification. ◀

Theorem 6 proves that the coinductive characterisation implies the standard CSP refinement-based notion of deadlock freedom. We have also proven the inverse direction of the implication in Isabelle/HOL, establishing the following equivalence.

▶ **Theorem 8** (Equivalence of Deadlock-Freedom Characterisations). *For any process P,*

$$\mathit{deadlock\_free}\ P \iff \mathrm{DFI}^{\checkmark}_{\Sigma}(P) \sqsubseteq P$$

This shows that our coinductive characterisation does not lose generality, while still providing a proof principle that is more amenable to automated reasoning.

6 Mechanised Deadlock Analysis in Isabelle/HOL

This section introduces the automated verification workflow for establishing deadlock freedom of the transition behaviour of RoboChart state machines in Isabelle/HOL using the running example. § 6.1 presents the workflow, and § 6.2 and § 6.3 present the two proof methods.

Table 1 Core normalisation laws for core CSP constructs.

ID	Rule
Seq-Pre	$a \rightarrow P; Q = a \rightarrow (P; Q)$
Seq-Inp	$c?a \in A \rightarrow P(a); Q = c?a \in A \rightarrow (P(a); Q)$
Bin-GEC	$b \;\&\; P \;\Box\; c \;\&\; Q \;=\; \Box_{i \in \{0,1\}} \big((\text{if } i = 0 \text{ then } b \text{ else } c) \;\&\; (\text{if } i = 0 \text{ then } P \text{ else } Q)\big)$
GEC-Norm	$\Big(\Box_{i \in \{0 \ldots n\}} b(i) \;\&\; P(i)\Big) \;\Box\; c \;\&\; Q =$ $\Box_{i \in \{0 \ldots n+1\}} \big((\text{if } i \leq n \text{ then } b(i) \text{ else } c) \;\&\; (\text{if } i \leq n \text{ then } P(i) \text{ else } Q)\big)$
Int-Seq	$(P \triangle Q); R = P \triangle (Q; R)$ if P does not terminate, Q does not initially.

6.1 Overview of the workflow

To analyse a RoboChart model for deadlock freedom, we propose the steps below.

1. *Model construction in RoboTool.* This involves defining a RoboChart state machine in RoboTool, optionally supplying invariants over shared variables.
2. *Translation to HOL-CSP.* RoboTool generates a *Circus* enriched semantics, and then produces an Isabelle theory in HOL-CSP syntax (as that in Listing 2). A suitable invariant is identified manually for the assume-guarantee reasoning, but enriched automatically.
3. *Loading rules in Isabelle.* The generated theory is loaded together with HOL-CSP and our libraries of normalisation and reduction rules shown later, providing the algebraic backbone of the automation that is achieved in the next step.
4. *Automated deadlock checking.* We developed the proof method `deadlock_free` to discharge the proof obligation with provided invariant, and `find_counterexample` to generate a counterexample leading to a potential deadlock when the automated proof fails.

6.2 Automated proof method: deadlock_free

The `deadlock_free` method provides a push-button procedure for processes in the prototypical Trans form (Definition 3). The method is built upon the normalisation and the reduction rules shown in Tables 1 and 2, as well as the HOL-CSP library, and proceeds as follows.

Step 1. Application of the coinduction theorem. We use Theorem 6 to rewrite the goal `deadlock_free`$(\sqcap_{n \in V} Trans(n))$ into a refinement obligation universally quantified over a single parameter n. The proof obligation is reduced to showing that for any arbitrary index $n \in V$, the refinement holds. In the theorem, the term $Q(x)$ is instantiated with the defining body of $Trans(n)$ following the structure of Definition 3. For the running example, the term $Q(x)$ is instantiated with the definition of `Trans` in Listing 2, whose recursive body starts at line 9, and the refinement is universally quantified over the active node parameter n.

Step 2. Case analysis on the active node. The parameter `n` is subject to a case analysis, yielding one subgoal per node. Each subgoal is concerned with the outgoing transitions of a single node, matching the per-node requirement of Definition 2 ("for every non-terminal node n, there exists an enabled outgoing transition ..."). In the running example, case analysis over `n` produces five subgoals, one for each node of the state machine, as shown in Fig. 2.

For instance, the subgoal for the state `NID_move` covers two branches of the external choice, corresponding to the two outgoing transitions (lines 14 and 15). The subgoal for the state `NID_stop` is for only the branch on line 13 of Listing 2, and its body has the shape below.

$$\mathrm{DFI}_{\Sigma}^{\checkmark}\left(\sqcap_{n\in V} Trans(n)\right) \sqsubseteq \texttt{SSTOP}\triangle\texttt{(get\_obs?obs} \to \texttt{(get\_pos?pos} \to$$
$$\texttt{assume ((obs = 1} \vee \texttt{ obs = -1)} \wedge \texttt{pos <= dest)(Trans.NID\_stop)}$$
$$\texttt{((internal\_.NID\_stop} \to \texttt{Skip);(SSTOP}\triangle\texttt{(exit -> Skip));}$$
$$\texttt{SSTOP}\triangle\texttt{((exited} \to \texttt{Skip);(enter\_f0} \to \texttt{Trans.NID\_f0}))))).$$

Step 3. Introduction of state variables. The outer `SSTOP` interrupt of Definition 3 has no impact on the deadlock freedom of the model, and our method uses the SSTOP-Red rule in Table 2 to eliminate it. The remaining variable-input prefix $get_{\vec{x}}?\vec{v} \to \cdots$ is then eliminated using Inp-Red rule in Table 2, introducing a universal quantification over $\vec{v}$. Each subgoal thus takes the desired logical form "for every state context (valuation)" $\vec{v}$ from Definition 2. For the running example, the subgoal for the state `NID_stop` becomes as follows.

$$\forall\, obs\, pos.\ \mathrm{DFI}_{\Sigma}^{*\checkmark}\left(\sqcap_{n\in V} Trans(n)\right) \sqsubseteq \texttt{assume((obs = 1} \vee \texttt{ obs = -1)} \wedge \texttt{pos <= dest)}$$
$$\texttt{(Trans.NID\_stop)((internal\_.NID\_stop} \to \texttt{Skip);(SSTOP}\triangle\texttt{(exit} \to$$
$$\texttt{Skip));SSTOP}\triangle\texttt{((exited} \to \texttt{Skip);(enter\_f0} \to \texttt{Trans.NID\_f0}))))).$$

Step 4. Introduction of the invariant. Each branch contains the `assume` I wrapper from Definition 3. We unfold its definition and apply the Asm-Red rule in Table 2, which adds I to the list of hypotheses used in the refinement proof. Under the invariant I, the fallback branch of `assume` is unreachable, so the process behaves as its main continuation. The result is precisely the invariant-assumption indicated by Definition 2 ("a context satisfying I").

A `guar` I wrapper, when present, is handled by a case split on I. When I holds, `guar` I behaves as its continuation, and otherwise as `STOP`, by the definition of `guar`. This captures the invariant-preservation obligation from Definition 2 ("the resulting state context also satisfies I"). In the example, after this step, the goal for the state `NID_stop` is as follows.

$$I \implies \mathrm{DFI}_{\Sigma}^{*\checkmark}\left(\sqcap_{n\in V} Trans(n)\right) \sqsubseteq \texttt{(internal\_.NID\_stop} \to \texttt{Skip);(SSTOP}\triangle\texttt{(exit} \to$$
$$\texttt{Skip));SSTOP}\triangle\texttt{((exited} \to \texttt{Skip);(enter\_f0} \to \texttt{Trans.NID\_f0}))))).$$

Step 5. Normalisation to canonical form. Each branch now defines purely the transition behaviour as prescribed by Definition 3. We transform those branches to the normal form defined below, using the laws in Table 1, to make explicit the structure relevant to the enabledness and stepwise execution of transitions.

▶ **Definition 9** (Normal Form). *For a node n, the behaviour of its outgoing transitions $\mathcal{T}(n)$ is normalised if, and only if, it is defined in the form*

$$\sqcap_{t\in\mathcal{T}(n)} t.guard \,\&\, t.trigger \to P(t),$$
$$\textit{with } P(t) = \texttt{SSTOP}\triangle(\alpha_1 \to (\texttt{SSTOP}\triangle(\alpha_2 \to \cdots \to \texttt{Trans}(t.tgt)))).$$

The normal form is obtained by exhaustively applying a set of standard normalisation rules. Sequential composition is pushed inside prefixes; guarded choices are flattened into a single indexed external choice; and interrupts are propagated inside prefixes.

Table 2 Core reduction rules used in the proof development.

ID	Rule
Pre-Red	$\dfrac{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (P) \sqsubseteq Q}{\Gamma \vdash \mathrm{DFI}_\Sigma^{\checkmark}\ (P) \sqsubseteq a \to Q} \qquad \dfrac{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (P) \sqsubseteq Q}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (P) \sqsubseteq a \to Q}$
Inp-Red	$\dfrac{\Gamma \vdash \forall v.\ \mathrm{DFI}_\Sigma^{*\checkmark}\ (P) \sqsubseteq Q(v)}{\Gamma \vdash \mathrm{DFI}_\Sigma^{\checkmark}\ (P) \sqsubseteq a?x \to Q(x)}$
ICho-Red	$\dfrac{\Gamma \vdash a \in A}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (\sqcap\ v \in A.P\,v) \sqsubseteq P\,a}$
Asm-Red	$\dfrac{\Gamma, I \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (Q) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (Q) \sqsubseteq \mathit{assume}\,I\,Q\,P}$
Guar-Red	$\dfrac{\Gamma \vdash I \quad \Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (Q) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (Q) \sqsubseteq \mathit{guar}\,I\,P}$
GEC-Red	$\dfrac{\Gamma \vdash \exists i \in I.\ b(i) \quad \Gamma \vdash \forall i \in I.\ b(i) \Rightarrow S \sqsubseteq P(i)}{\Gamma \vdash S \sqsubseteq \underset{i \in I}{\square}\ b(i)\,\&\,P(i)}$
Skip-Red	$\dfrac{}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (P) \sqsubseteq \mathsf{Skip}}$
SSTOP-Red	$\dfrac{\Gamma \vdash \mathrm{DFI}_\Sigma^{\checkmark}\ (X) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}_\Sigma^{\checkmark}\ (X) \sqsubseteq \mathsf{SSTOP} \triangle P} \qquad \dfrac{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (X) \sqsubseteq P}{\Gamma \vdash \mathrm{DFI}_\Sigma^{*\checkmark}\ (X) \sqsubseteq \mathsf{SSTOP} \triangle P}$

For the running example, the subgoal for `NID_stop` becomes

$$\mathrm{DFI}_\Sigma^{*\checkmark}\left(\underset{n \in V}{\sqcap} \mathit{Trans}(n)\right) \sqsubseteq \texttt{internal\_.NID\_stop} \to \texttt{SSTOP} \triangle$$
$$\texttt{(exit} \to \texttt{SSTOP} \triangle \texttt{(exited} \to \texttt{enter\_f0} \to \texttt{Trans.NID\_f0))}.$$

Step 6. Refinement-based reduction. The reduction rules in Table 2 eliminate the prefixes, interrupts, and external-choice structure of the normalised process, leaving a goal of the form $\mathrm{DFI}_\Sigma^{*\checkmark}\ (\sqcap_{n \in V} \mathit{Trans}(n)) \sqsubseteq \mathit{Trans}(s)$, which is discharged using the rule ICho-Red to eliminate internal choice. Any simple guard or arithmetic side-conditions that may arise are discharged automatically by `simp`, `auto`, or Sledgehammer. We repeat these six steps for each state-specific subgoal, which collectively establishes the deadlock freedom of the entire transition system. For the running example, all five state subgoals are discharged, thereby proving the deadlock freedom of $\sqcap_{n \in V} \mathit{Trans}(n)$, that is, of the transition behaviour.

6.3 Counterexample search method: find_counterexample

When `deadlock_free` fails, `find_counterexample` helps to identify the cause by transforming the refinement obligation into a form suitable for Isabelle's model finders.

Unlike `deadlock_free`, which verifies nodes individually via case analysis (Step 2 in 6.2), this method keeps the state space unified for global counterexample search. Step 1 is identical to that in 6.2; subsequent steps differ as follows.

Step 2. Introduction of state variables and the invariant. Following the approach of Steps 3 and 4 in Section 6.2, we apply Inp-Red to introduce the state variables $\vec{v}$ via universal quantifiers and unfold the `assume` I wrapper to set $I(\vec{v})$ as a hypothesis. For our running example, this yields a global subgoal quantified over (n, obs, pos) under I, alongside a fallback goal for $\neg I$. This unified context allows model finders to explore all nodes simultaneously.

Step 3. Normalisation to indexed trees. For the first subgoal from Step 2, we apply Bin-GEC and GEC-Norm to flatten the transition structure into a single indexed choice: $\square_{i \in I}(g_i \ \& \ a_i)$. This recursively packs all guards into a nested `if-then-else` tree. The resulting subgoal presents a large guarded choice where the left-hand side of the guarded command operator (&) is a conditional branch selecting the enabling predicate for index i, and the right-hand side is a matching conditional branch selecting the corresponding process. This new subgoal is a structured "lookup table" that maps the current state to its enabling conditions.

Step 4. Reduction and search. Finally, GEC-Red reduces the first subgoal into logical predicates, among which the first goal is to prove that at least one transition is enabled, that is,$\forall \vec{v}.I(\vec{v}) \Longrightarrow \bigvee_{i \in I} \text{Guard}_i$, or in Isabelle, $\forall \vec{v}.I(\vec{v}) \Longrightarrow \exists i \in I. \bigwedge_{j \in I}(i = j \longrightarrow \text{Guard}_j)$. With this, we enable Isabelle's *nitpick* or *quickcheck* to efficiently find a valuation of $(n, \vec{v})$ where I holds, but all guards are false, pinpointing the cause of a potential deadlock.

7 Case studies and evaluation

We have applied our verification strategy to four case studies, including our running example, as summarised below. All experiments were conducted on a laptop with an Intel Core i9-14900HX processor, 128GB RAM, running Ubuntu Linux.

Running example. This example (Fig. 2) models an obstacle-avoiding controller, where the system behaviour is driven by shared variables. Although the state space is relatively small, it captures the challenge of shared variables: the sensor reading(`obs`) and the position(`pos`) are potentially accessible by other concurrent processes, meaning their values could change externally. By enrichment with our invariant of `((obs=1 ∨ obs=-1) ∧ pos ≤ dest)`, the `deadlock_free` method automatically proves the property:

```
lemma Trans_ObsDet_ddlf: "deadlock_free (⊓ n ∈ UNIV. Trans n)"
  apply (deadlock_free P_def: Trans.simps)
  by blast+
```

We also evaluated the proof with a trivial invariant (`True`), which failed as expected. Finally, in the unenriched configuration, the proof also failed. In both failed cases, `find_counterexample` identified a concrete deadlock (e.g., at node `NID_detect` with `obs=3`), helping the user refine the system's assumptions.

Adaptive navigation for a TurtleBot 4 [8]. This case study implements a form of the MAPE-K loop [20], which extends the TurtleBot 4's[1] navigation capabilities to handle unanticipated sensor failures. We focus on a RoboChart MakePlan state machine (defining the Plan (P)

[1] `https://clearpathrobotics.com/turtlebot-4/`

phase of MAPE-K). The machine computes rotation strategies based on sensor inputs and system constraints. It contains 9 states and 13 transitions and requires an invariant that the minimum rotation rate must not exceed the maximum one, $\mathsf{minRotation} \leq \mathsf{maxRotation}$. Deadlocks may arise when the transition guards do not cover all possible combinations of these parameters. We first applied `deadlock_free`, but the goal was not discharged. This indicated a potential deadlock, so we used `find_counterexample`, which produced the valuation $\mathsf{minRotation} = -1$ and $\mathsf{maxRotation} = 0$. This identifies a state context from which no transition is enabled. The transition conditions were then revised to ensure that all valuations are properly covered by the guards. Reapplying `deadlock_free` then succeeded, establishing that the enriched MakePlan is deadlock-free.

We attempted to prove deadlock freedom for the corrected MakePlan without invariant enrichment. However, the corrected MakePlan can deadlock. So, the proof fails, and we get a counterexample. This case study highlights the complementary roles of the two automated methods: `deadlock_free` provides push-button verification for correct models, while `find_counterexample` offers counterexamples for incorrect ones. Together, they provide a practical, mechanised workflow for verifying deadlock freedom of RoboChart transition behaviour with shared-variable concurrency.

Autonomous underwater vehicle (AUV) [15]. The AUV is an industrial robot for subsea maintenance, operating in both autonomous and human-controlled modes. Its primary safety challenge is avoiding collisions with subsea infrastructure under either mode of operation. A "Last Response Engine" (LRE) safety monitoring component was developed by an industry partner to ensure the vehicle operates safely. The LRE manages four operating modes determined by safety conditions: (i) Operator Control Mode; (ii) Main Operating Mode, for automatic operation under safe conditions; (iii) High Caution Mode, for reduced-speed operation under elevated risk; and (iv) Collision Avoidance Mode, for emergency manoeuvring under high risk. The state machine of LRE has 5 states and 18 transitions. We proved deadlock freedom of the transition behaviour of the LRE safety monitoring component. This case study involves state variables of type real, leading to an infinite state space. Our proof method handles this symbolically and does not rely on state enumeration or finiteness assumptions. We also attempted verification using FDR on the machine described above, but it ran out of memory. To approximate real-valued variables in FDR, we used five integers; even then, the state space remained too large. In contrast, our approach handles real-valued variables natively in Isabelle, and the proof completes in 4s elapsed time on average.

Chemical Detector Robot [23]. This case study considers an autonomous robot tasked with identifying hazardous chemical sources while avoiding obstacles. The system coordinates multiple concurrent processes: while one component executes a random-walk trajectory, a separate gas-analysis component continuously monitors sensor streams. Upon detecting an intensity above a safety threshold, the analysis logic must interrupt the movement cycle to drop a marker and initiate a stop. This model involves mixed synchronous-asynchronous communication. Our `deadlock_free` method establishes deadlock freedom for the transition behaviour of the gas-analysis state machine, which has 6 states and 7 transitions. We have an invariant stating that the sensor reading is never empty and assumes a default value when no gas is present. This invariant ensures that the sensor pipeline remains productive. The method decomposes the verification into subgoals for each state and proves them automatically, with Sledgehammer discharging the remaining arithmetic side-conditions.

Summary. Across the running example and the three case studies, deadlock freedom of the transition behaviour is established automatically using the `deadlock_free` method when suitable invariants are provided. For the running example and the TurtleBot case study, we evaluated the proofs both with and without invariants. When suitable invariants are provided, the proofs succeed automatically; without them, the verification fails and counterexamples are generated. The AUV and Chemical Detector models are verified using the default invariant `True`, since their transition guards and control structure are sufficient to ensure progress, without requiring additional state assumptions. The chemical detector case shows that, after the structural decomposition is handled by our method, certain arithmetic side conditions may require support from Sledgehammer to complete the proof.

For each case study, we measured the elapsed time of applying the `deadlock_free` method, restarting Isabelle between runs to avoid caching effects. Over ten runs per case study, the average time per proof was within the range of 0.4s to 4s with an average of 1.4s. In contrast, with FDR, the verification time for the more complex models, such as the TurtleBot, can be slower by a factor of 100 compared to verification in Isabelle, due to the need to explore each state explicitly. This substantiates our claim that theorem proving gives a tangible benefit to verification scalability. All the RoboChart models, HOL-CSP models, proofs, and time measurements are available online. All verification results are obtained within Isabelle/HOL and are therefore backed by fully checked proofs in the trusted kernel. While our approach emphasises automation, the generated proof obligations, invariants, and intermediate reasoning steps remain accessible to the user and can be inspected within Isabelle. In this sense, the framework does not rely on a black-box decision procedure but produces machine-checked evidence for all verification results.

8 Related work

Deadlock freedom is a key property widely studied in the literature. We consider it in the context of CSP-style concurrency. FDR checks deadlock freedom via explicit state exploration, requiring a finite transition relation. For example, Antonino et al. [3] worked on the compositional verification of deadlock-freedom of CSP processes using FDR. However, real-valued variables introduce an infinite state space, making such enumeration infeasible.

By contrast, HOL-CSP [10, 35] directly implements the set-based failures-divergences semantics in Isabelle, and allows symbolic reasoning about the transition relation. We have extended the HOL-CSP library by introducing a coinductive specification for deadlock freedom, supported by a set of formal normalisation and reduction laws. By leveraging the symbolic reasoning capabilities of HOL-CSP, this extension enables automated verification for systems with infinite state spaces, effectively overcoming the limitations of explicit-state enumeration as illustrated by our use cases. CSP-Prover [18] provides a deep embedding of CSP in Isabelle/HOL and supports refinement proofs using algebraic laws and induction. While it enables reasoning about scalable concurrent systems, proofs are semi-automatic and require significant user interaction. Our work complements this line of research by providing a more automated verification approach for deadlock freedom in CSP-style models.

Foster et al. [13] have presented a mechanised reactive design semantics in Isabelle/ UTP and developed proof tactics for verifying safety and liveness properties of RoboChart state machines. In that work, the authors use a sequentialised interpretation of state-machine behaviour, simplifying the reasoning process. Yan et al. [38] have applied the Z-Machine notation to compositional verification of RoboChart state machines. Those works have demonstrated that invariant-based reasoning can be successfully automated in Isabelle, but

their scope does not address compositional reasoning in the presence of shared variables. Our work provides a foundation for compositional reasoning in the presence of shared variables via a coinductive specification of deadlock freedom and an assume–guarantee strategy, and also provides a basis for extending the approach to concurrent systems.

Oliveira et al. [28] provided composition patterns that preserve deadlock freedom by construction. That work has been extended to UML-based communicating components [11], and is complementary to ours, since we focus on a posteriori component-level deadlock freedom. That work can perhaps be applied to demonstrate that the composition of two deadlock-free RoboChart controllers with only asynchronous (that is, buffered) connections is deadlock-free. It cannot, however, deal with arbitrary connections, as we seek to handle.

Beyond work on the verification of RoboChart models, we consider a broader body of research on the automated verification of statecharts using theorem-proving approaches. Helke et al. [17] presented an approach for verifying statecharts with infinite data spaces. Their method uses Isabelle/HOL to construct an abstract model via data abstraction, and applies model checking (e.g. SMV) to verify temporal properties. Our approach focuses on deductive reasoning about deadlock freedom within the theorem prover.

A survey [2] provides a comprehensive overview of work from 1997 to 2021 on formalising UML state machine semantics, but largely aimed at enabling model checking at the design stage. It also discusses translations to formal frameworks such as the Prototype Verification System (PVS) [30], KIV [7], B [1], and Z [34], enabling, in principle, verification using deductive techniques. However, among the works surveyed, only the KIV-based work of Balser et al. [6] substantially addresses verification using theorem proving. The others focus on semantic encoding, translation, or model-checking. We therefore discuss Balser et al.'s work, together with other related approaches, to better position our contribution.

Balser et al. [6] present an interactive approach to verifying UML state machines, using symbolic execution and induction in KIV to prove temporal properties. While capable of handling infinite-state systems, it relies on user-guided proofs and offers limited automation.

A further line of work translates UML models into formal methods such as B or Event-B. Lano et al. [22] define a translation from UML class diagrams into B, which allows checking consistency, verifying invariants, and analysing model behaviour via animation. However, their work focuses on the translation and the range of properties that can be verified, rather than on developing automated verification techniques.

Snook et al. [33] focus on translating UML class diagrams into Event-B to enable invariant-based verification through refinement. The work does not address the development of proof methods for behavioural properties such as deadlock freedom.

Morris et al. [26] translate State Chart XML (SCXML) with run-to-completion semantics into Event-B, enabling invariant-based verification via refinement. Safety properties are verified automatically in the Event-B/Rodin framework by discharging proof obligations using theorem provers, AtelierB, and SMT solvers. This automation relies on structured modelling and refinement, which yields proof obligations of a predictable form, but depends on careful invariant design and is closely tied to invariant preservation rather than specialised behavioural proof methods like we do. Deadlock freedom is not handled with the same level of automation, but is informally argued by showing that execution steps always make progress and eventually terminate under standard fairness assumptions.

Taken together, these works suggest that, outside of the RoboChart setting studied here, automated verification of deadlock freedom for state machine or statechart models using theorem-proving approaches remains largely unexplored.

9 Conclusions and future work

We have introduced a theorem-proving method for verifying deadlock freedom and invariant properties of CSP models with shared variables in Isabelle/HOL. The method combines a coinductive characterisation of deadlock freedom with assume–guarantee reasoning, and is supported by a mechanised shallow embedding of CSP. The technique is generally applicable to CSP processes that follow similar shared-variable interaction patterns, with RoboChart providing a rich and realistic case study that demonstrates its effectiveness. Once users provide invariants during the modelling phase, verification itself is automated. Crucially, invariants are considered part of the design assumptions, not the verification procedure. For adaptive systems where invariants remain valid during runtime reconfiguration, our technique enables automated runtime verification without manual intervention.

Our method complements model checking by avoiding state-space explosion via symbolic reasoning, and advances Isabelle-based verification by supporting compositional reasoning for systems with shared variables. Our results pave the way for scalable verification of concurrent systems with large state spaces and even unbounded data.

Future work will extend the verification from transition behaviour to RoboChart state machines, controllers, and modules, considering deadlock freedom and other refinement properties. For these components, we currently adopt a strategy of structural abstraction and instantiation. For example, we can formulate abstract definitions of nodes and memory processes directly in HOL-CSP within Isabelle.

While reasoning about parallel composition in CSP is not the focus of this work, the underlying HOL-CSP framework already provides algebraic laws that enable such reasoning. In particular, external choice distributes over parallel composition:

$$\begin{aligned}\left(\Box_{a\in A}\, a \to P(a)\right) \;|[S]|\; \left(\Box_{b\in B}\, b \to Q(b)\right) = {} & \left(\Box_{a\in A\setminus S}\, a \to (P(a) \;|[S]|\; (\Box_{b\in B}\, b \to Q(b)))\right) \\ & \Box \left(\Box_{b\in B\setminus S}\, b \to ((\Box_{a\in A}\, a \to P(a)) \;|[S]|\; Q(b))\right) \\ & \Box \left(\Box_{c\in A\cap B\cap S}\, x \to (P(c) \;|[S]|\; Q(c))\right)\end{aligned}$$

This law, together with the coinduction principles in this paper, provides a formal basis for extending our approach to parallel composition, the standard mechanism for modelling concurrency in CSP, enabling compositional reasoning about concurrent processes.

We plan to lift assume–guarantee reasoning to system-level verification of interacting components, and to evaluate the approach on larger CSP-based models. In addition, the UTP-based semantics of *Circus* opens opportunities for integrating physical environment models, following recent Isabelle-based work on cyber-physical systems.

References

1 Jean-Raymond Abrial. The B-book. *Assigning Programs to Meaning, to be published*, 1996.

2 Étienne André, Shuang Liu, Yang Liu, Christine Choppy, Jun Sun, and Jin Song Dong. Formalizing UML state machines for automated verification–A survey. *ACM Computing Surveys*, 55(13s):1–47, 2023. doi:10.1145/3579821.

3 Pedro Antonino, Augusto Sampaio, and Jim Woodcock. A refinement based strategy for local deadlock analysis of networks of csp processes. In *International Symposium on Formal Methods*, pages 62–77. Springer, 2014. doi:10.1007/978-3-319-06410-9_5.

4 Benoît Ballenghien, Safouan Taha, and Burkhart Wolff. HOL-CSPM - Architectural operators for HOL-CSP. *Archive of Formal Proofs*, December 2023. , Formal proof development. URL: https://isa-afp.org/entries/HOL-CSPM.html.

5 Benoît Ballenghien, Safouan Taha, Burkhart Wolff, and Lina Ye. HOL-CSP Version 2.0. *Archive of Formal Proofs*, April 2019. , Formal proof development. URL: https://isa-afp.org/entries/HOL-CSP.html.

6 Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive verification of UML state machines. In *International Conference on Formal Engineering Methods*, pages 434–448. Springer, 2004. doi:10.1007/978-3-540-30482-1_36.

7 Michael Balser, Wolfgang Reif, Gerhard Schellhorn, and Kurt Stenzel. KIV 3.0 for provably correct systems. In *International Workshop on Current Trends in Applied Formal Methods*, pages 330–337. Springer, 1998. doi:10.1007/3-540-48257-1_23.

8 James Baxter, Bert Van Acker, Morten Kristensen, Thomas Wright, Ana Cavalcanti, and Cláudio Gomes. Formal architectural patterns for adaptive robotic software. In *International Conference on Fundamental Approaches to Software Engineering*, pages 145–165. Springer Nature Switzerland Cham, 2025.

9 Ana Cavalcanti, Madiel Conserva Filho, Pedro Ribeiro, and Augusto Sampaio. Laws of timed state machines. *The Computer Journal*, 67(6):2066–2107, 2024. doi:10.1093/COMJNL/BXAD124.

10 P. Crisafulli, S. Taha, and B. Wolff. Modeling and analysing cyber-physical systems in HOL-CSP. *Robotics and Autonomous Systems*, 170, 2023.

11 F. Falcao, L. Lima, A. Sampaio, and P. Antonino. A formal component model for UML based on CSP aiming at compositional verification. *Software and Systems Modeling*, 2024.

12 Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/circus: A process specification and verification environment. In *International Conference on Verified Software: Tools, Theories, Experiments*, pages 243–260. Springer, 2012. doi:10.1007/978-3-642-27705-4_20.

13 Simon Foster, James Baxter, Ana Cavalcanti, Alvaro Miyazawa, and Jim Woodcock. Automating verification of state machines with reactive designs and Isabelle/UTP. In *International Conference on Formal Aspects of Component Software*, pages 137–155. Springer, 2018. doi:10.1007/978-3-030-02146-7_7.

14 Simon Foster, Chung-Kil Hur, and Jim Woodcock. Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL. *arXiv preprint arXiv:2105.05133*, 2021. arXiv:2105.05133.

15 Simon Foster, Yakoub Nemouchi, Colin O'Halloran, Karen Stephenson, and Nick Tudor. Formal model-based assurance cases in Isabelle/SACM: An autonomous underwater vehicle case study. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*, pages 11–21, 2020. doi:10.1145/3372020.3391559.

16 T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 187–201, 2014.

17 Steffen Helke and Florian Kammuller. Verification of statecharts using data abstraction. *International Journal of Advanced Computer Science and Applications*, 7(1), 2016.

18 Yoshinao Isobe and Markus Roggenbach. CSP-Prover—A proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010. doi:10.11185/IMT.5.32.

19 Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983. doi:10.1145/69575.69577.

20 Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003. doi:10.1109/MC.2003.1160055.

21 Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. The epsilon object language (EOL). In *European conference on model driven architecture-foundations and applications*, pages 128–142. Springer, 2006.

22 Kevin Lano, David Clark, and Kelly Androutsopoulos. UML to B: Formal verification of object-oriented models. In *International Conference on Integrated Formal Methods*, pages 187–206. Springer, 2004. doi:10.1007/978-3-540-24756-2_11.

23 Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, and Jon Timmis. Automatic property checking of robotic applications. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3869–3876, 2017. doi:10.1109/IROS.2017.8206238.

24 Alvaro Miyazawa, Pedro Ribeiro, Wei Li, Ana Cavalcanti, Jon Timmis, and Jim Woodcock. RoboChart: modelling and verification of the functional behaviour of robotic applications. *Software and Systems Modeling*, 18(5):3097–3149, 2019. doi:10.1007/s10270-018-00710-z.

25 Alvaro Miyazawa, Pedro Ribeiro, Kangfeng Ye, Ana Cavalcanti, Wei Li, Jim Woodcock, and Jon Timmis. Robochart reference manual, 2021. URL: https://robostar.cs.york.ac.uk/publications/techreports/reports/robochart-reference.pdf.

26 Karla Morris, Colin Snook, Thai Son Hoang, Geoffrey Hulette, Robert Armstrong, and Michael Butler. Formal verification and validation of run-to-completion style state charts using Event-B. *Innovations in Systems and Software Engineering*, 18(4):523–541, 2022. doi:10.1007/S11334-021-00416-4.

27 Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002. doi:10.1007/3-540-45949-9.

28 M. V. M. Oliveira, P. Antonino, R. Ramos, A. Sampaio, A. Mota, and A. W. Roscoe. Rigorous development of component-based systems using component metadata and patterns. *Formal Aspects of Computing*, 2016.

29 Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A UTP semantics for Circus. *Formal Aspects of Computing*, 21(1):3–32, 2009. doi:10.1007/S00165-007-0052-5.

30 Sam Owre, John M Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

31 A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

32 Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. The Epsilon Generation Language. In *European conference on model driven architecture-foundations and applications*, pages 1–16. Springer, 2008.

33 Colin Snook, Vitaly Savicks, and Michael Butler. Verification of uml models by translation to uml-b. In *International Symposium on Formal Methods for Components and Objects*, pages 251–266. Springer, 2010. doi:10.1007/978-3-642-25271-6_13.

34 J Michael Spivey and Jean-Raymond Abrial. *The Z notation*, volume 29. Prentice Hall Hemel Hempstead, 1992.

35 S. Taha, B. Wolff, and L. Ye. Philosophers may dine — definitively! In *Proc. 16th Intl. Conf. on Integrated Formal Methods*, LNCS. Springer, 2020. doi:10.1007/978-3-030-63461-2_23.

36 Jim Woodcock, Ana Cavalcanti, Simon Foster, Marcel Oliveira, Augusto Sampaio, and Frank Zeyda. Utp, circus, and isabelle. In *Theories of Programming and Formal Methods: Essays Dedicated to Jifeng He on the Occasion of His 80th Birthday*, pages 19–51. Springer, 2023. doi:10.1007/978-3-031-40436-8_2.

37 Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.

38 Fang Yan, Simon Foster, and Ibrahim Habli. Automated compositional verification for robotic state machines using Isabelle/HOL. In *2023 27th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 167–176. IEEE, 2023. doi:10.1109/ICECCS59891.2023.00029.