Automating Verification of State Machines with Reactive Designs and Isabelle/UTP

Simon Foster, James Baxter, Ana Cavalcanti,
Alvaro Miyazawa, and Jim Woodcock

University of York

Abstract. State-machine based notations are ubiquitous in the description of component systems, particularly in the robotic domain. To ensure these systems are safe and predictable, formal verification techniques are important, and can be cost-effective if they are both automated and scalable. In this paper, we present a verification approach for a diagrammatic state machine language that utilises theorem proving and a denotational semantics based on Unifying Theories of Programming (UTP). We provide the necessary theory to underpin state machines (including induction theorems for iterative processes), mechanise an action language for states and transitions, and use these to formalise the semantics. We then describe the verification approach, which supports infinite state systems, and exemplify it with a fully automated deadlock-freedom check. The work has been mechanised in our proof tool, Isabelle/UTP, and so also illustrates the use of UTP to build practical verification tools.

1 Introduction

The recent drive for adoption of autonomous robots into situations where they interact closely with humans means that such systems have become safety critical. To ensure that they are both predictable and safe within their applied context, it is important to adequately prototype them in a variety of scenarios. Whilst physical prototyping is valuable, there is a limit to the breadth of scenarios that can be considered. Thus, techniques that allow virtual prototyping, based on mathematically principled models, can greatly enhance the engineering process. In particular, formal verification techniques like model checking and theorem proving can enable exhaustive coverage of the state space.

Diagrammatic notations are widely applied in component modelling, particularly in the modelling of robotic controllers via state machines. Industry standards, like UML and SysML, provide languages that allow description of component interfaces, the architecture and connectivity, and the behaviour of individual components. These languages have proved popular, due to, on the one hand, accessibility and ease of use, and on the other hand, the provision of precise modelling techniques. In order to leverage formal verification techniques in this context there is a need for both formal semantics and automated tools. Since UML is highly extensible, a particular challenge is to provide semantic models that support extensions such as real-time, hybrid computation, and probability.

RoboChart [1, 23] is a diagrammatic language for the description of robotic controllers with denotational semantics based on Hoare and He's *Unifying Theories of Programming* [2] (UTP). The core of RoboChart is a formalised state machine notation that can be considered a subset of UML/SysML state machine diagrams enriched with time and probability constructs. Each state machine has a well defined interface describing the events that are externally visible. The behaviour of states and transitions is described using a formal action language that corresponds to a subset of the *Circus* modelling language [3]. The notation supports real-time constraints, through delays, timeouts and deadlines, and also probabilistic choices, to express uncertainty. The use of UTP, crucially, enables us to provide various semantic models for state machines that account for different computational paradigms, and yet are linked through a common foundation.

In previous work [1], model checking facilities for RoboChart have been developed and applied in verification. This provides a valuable automated technique for model development, that allows detection of problems during the early development stages. However, explicit state model checking is also limited to checking finite state models. In practice this means that data types must be abstracted with a small number of elements. Therefore, in order to exhaustively check the potentially very large or infinite state space of many robotic applications, symbolic techniques, like theorem proving, are required. In order for this to be practically applicable, like model checking, it is necessary to implement the denotational semantics so that automated proof tactics can be used.

In this paper we present a verification approach for state machines, based on a subset of RoboChart, that has been mechanised in the Isabelle/HOL [4] proof assistant. We mechanise the meta-model, including its data types and well-formedness constraints, and a proof tactic to validate the underlying graph. The meta-model utilises a reactive program theory with optimised proof support that we have mechanised in our UTP implementation, Isabelle/UTP [5]. We use reactive programs as a semantic domain for a dynamic semantics, based on Dijkstra's guarded iteration construct [20]. This semantics can be used to perform verification of infinite-state systems by theorem proving, with the help of a verified induction theorem, and Isabelle/HOL's automated proof facilities [13]. Our work is applicable to further mechanised UTP theories, and so can be extended to handle time [7], probability [11], and other paradigms. It also serves as a template for building verification tools with Isabelle/UTP.

In §2 we provide background material on RoboChart, UTP, Isabelle/HOL, and reactive designs. In §3 we begin our contributions by extending reactive designs with support for guarded iteration, and an induction theorem for proving invariants. In §4 we mechanise reactive programs in Isabelle/UTP, based on the reactive-design theory, and provide symbolic evaluation theorems. In §5 we mechanise a static semantics of state machines, including well-formedness checks. In §6 we provide the dynamic semantics, utilising the result from §3, and prove a specialised induction law. In §7 we outline our verification approach, and prove deadlock freedom for an example state machine that satisfies the restrictions of our semantic model. Finally, in §8 we conclude and highlight related work.

2 Preliminaries

In this section, we describe the background material for our work.

2.1 RoboChart

RoboChart [1] describes robotic systems in terms of a number of controllers that communicate using shared channels. Each controller has a well defined interface, and its behaviour is described by one or more state machines. A machine has local state variables and constants, and consists of nodes and transitions, with behaviour specified using a formal action language [3]. Advanced features such as hierarchy, shared variables, real-time constraints, and probability are supported.

A machine, GasAnalysis, is shown in Figure 1; we use it as a running example. It models a component of a chemical detector robot [6] that searches for dangerous chemicals using its spectrometer device, and drops flags at such locations. GasAnalysis is the component that decides how to respond to a sensor reading. If gas is detected, then an analysis is performed to see whether the gas is above or below a given threshold. If it is below, then the robot attempts to triangulate a position for the source location and turns toward it, and if it is above, it stops.

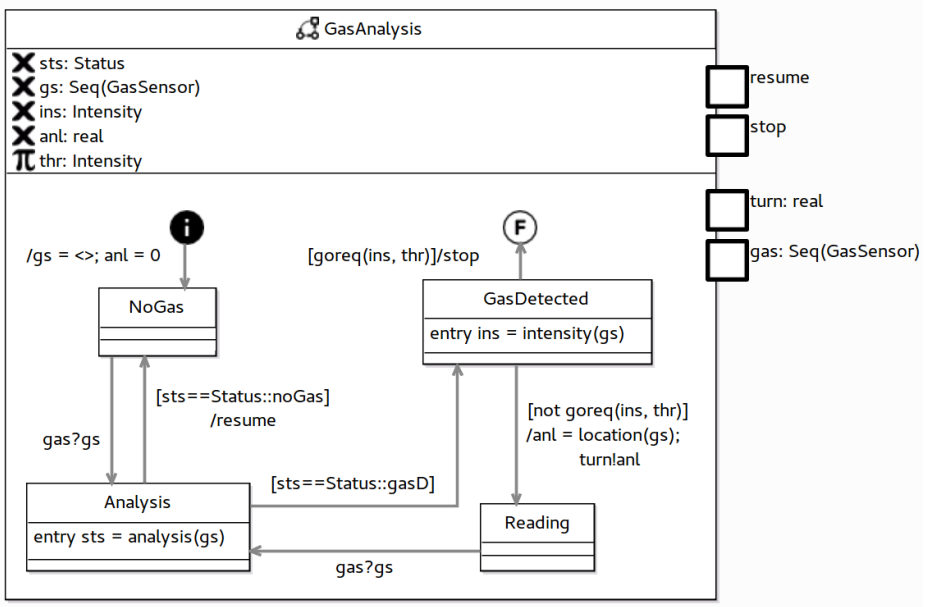

Fig. 1. GasAnalysis state machine in RoboChart

The interface consists of four events. The event *gas* is used to receive sensor readings, and *turn* is used to communicate a change of direction. The remaining events, *resume* and *stop* carry no data, and are used to communicate that the robot should resume its searching activities, or stop. The state machine uses four state variables: *sts* to store the gas analysis status, *gs* to store the

present reading, *ins* to store the reading intensity, and *anl* to store the angle the robot is pointing. It also has a constant *thr* to store the gas intensity threshold. RoboChart provides basic types for variables and constants, including integers, real numbers, sets, and sequences (*Seq(t)*). The user can also define additional types, that can be records, enumerations, or entirely abstract. For example, the type *Status* is an enumerated type with constructors *gasD* and *noGas*.

The behaviour is described by 6 nodes, including an initial node (i); a final node (F); and four states: *NoGas*, *Analysis*, *GasDetected*, and *Reading*. The transitions are decorated with expressions of the form *trigger*[*condition*]/*statement*. When the event *trigger* happens and the guard *condition* is true, then *statement* is executed executed, before transitioning to the next state. All three parts can optionally be omitted. RoboChart also permits states to have entry, during, and exit actions. In our example, both *Analysis* and *GasDetected* have entry actions.

The syntax of actions is given below, which assumes a context where event and state variable identifiers have been specified using the nonterminal *ID*.

Definition 2.1 (Action Language Syntax).

$$\begin{aligned} \mathit{Action} &:= \mathit{Event} \mid \textbf{\textit{skip}} \mid \mathit{ID} := \mathit{Expr} \mid \mathit{Action} \ ; \ \mathit{Action} \mid \\ &\quad\ \ \textbf{\textit{if}}\ \mathit{Expr}\ \textbf{\textit{then}}\ \mathit{Action}\ \textbf{\textit{else}}\ \mathit{Action}\ \textbf{\textit{end}} \\ \mathit{Event} &:= \mathit{ID} \mid \mathit{ID}\,\textbf{?}\,\mathit{ID} \mid \mathit{ID}\,\textbf{!}\,\mathit{Expr} \end{aligned}$$

An action is either an event, a ***skip***, an assignment, a sequential composition, or a conditional. An event is either a simple synchronisation on some identified event e, an input event ($e?x$) that populates a variable x, or an output event ($e!v$). We omit actions related to time and operations for now.

Modelling with RoboChart is supported by the Eclipse-based RoboTool[1], from which Figure 1 was captured. RoboTool automates verification via model checking using FDR4, and its extension to incorporate the verification approach presented here is ongoing work.

2.2 Unifying Theories of Programming

UTP is a framework for the formalisation of computational semantic domains that are used to give denotational semantics to a variety of programming and modelling languages. It employs alphabetised binary relations to model programs as predicates relating the initial values of variables (x) to their later values (x'). UTP divides variables into two classes: (1) program variables, that model data, and (2) observational variables, that encode additional semantic structure. For example, $\mathit{clock} : \mathbb{N}$ is a variable to record the passage of time. Unlike a program variable, it makes no sense to assign values to *clock*, as this would model arbitrary time travel. Therefore, observational variables are constrained using healthiness conditions, which are encoded as idempotent functions on predicates. For example, application of $\textbf{\textit{HT}}(P) \triangleq (P \wedge \mathit{clock} \leq \mathit{clock}')$ results in a healthy predicate that specifies there is no reverse time travel.

[1] https://www.cs.york.ac.uk/circus/RoboCalc/robotool/

The observational variables and healthiness conditions give rise to a subset of the alphabetised relations called a *UTP theory*, which can be used to justify the fundamental theorems of a computational paradigm. A UTP theory is the set of fixed-points of the healthiness condition: $[\![\boldsymbol{H}]\!] \triangleq \{P \mid \boldsymbol{H}(P) = P\}$. A set of signature operators can then be defined that are closed under the theory's healthiness conditions, and are thus guaranteed to construct programs that satisfy these theorems. UTP theories allow us to model a variety of paradigms beyond simple imperative programs, such as concurrency [2, 3], real-time [7], object orientation [8], hybrid [9, 10], and probabilistic systems [11].

The use of relational calculus means that the UTP lends itself to automated program verification using refinement $S \sqsubseteq P$: program P satisfies specification S. Since both S and P are specified in formal logic, and refinement equates to reverse implication, we can utilise interactive and automated theorem proving technology for verification. This allows application of tools like Isabelle/HOL to program verification, which is the goal of our tool, Isabelle/UTP [5].

2.3 Isabelle/HOL and Isabelle/UTP

Isabelle/HOL [4] consists of the Pure meta-logic, and the HOL object logic. Pure provides a term language, polymorphic type system, syntax translation framework for extensible parsing and pretty printing, and an inference engine. The jEdit-based IDE allows LaTeX-like term rendering using Unicode. An Isabelle theory consists of type declarations, definitions, and theorems, which are usually proved by composition of existing theorems. Theorems have the form of $[\![P_1;\ \cdots\ ;\ P_n]\!] \Longrightarrow Q$, where P_i is an assumption, and Q is the conclusion. The simplifier tactic, simp, rewrites terms using theorems of the form $f(x_1 \cdots x_n) \equiv y$.

HOL implements an ML-like functional programming language founded on an axiomatic set theory similar to ZFC. HOL is purely definitional: mathematical libraries are constructed purely by application of the foundational axioms, which provides a highly principled framework. HOL provides inductive datatypes, recursive functions, and records. Several basic types are provided, including sets, functions, numbers, and lists. Parametric types are written by precomposing the type name, τ, with the type variables $[a_1, \cdots, a_n]\tau$, for example $[nat]list$[2].

Isabelle/UTP [5, 12] is a semantic embedding of UTP into HOL, including a formalisation of the relational calculus, fundamental laws, proof tactics, and facilities for UTP theory engineering. The relational calculus is constructed such that properties can be recast as HOL predicates, and then automated tactics, such as auto, and sledgehammer [13], can be applied. This strategy is employed by our workhorse tactic, rel-auto, which automates proof of relational conjectures.

Proof automation is facilitated by encoding variables as lenses [5]. A lens $x :: \tau \Longrightarrow \alpha$ characterises a τ-shaped region of the type α using two functions: $\mathit{get}_x :: \alpha \to \tau$ and $\mathit{put}_x :: \alpha \to \tau \to \alpha$, that query and update the region, respectively. Intuitively, x is a variable of type τ within the alphabet type α. Alphabet types can be encoded using the ***alphabet*** $r = f_1 :: \tau_1 \cdots f_n :: \tau_n$ command, that

[2] The square brackets are not used in Isabelle; we add them for readability.

constructs a new record type r with n fields, and a lens for each field. Lenses can be independent, meaning they cover disjoint regions, written $x \bowtie y$, or contained within another, written $x \preceq y$. These allow us to express meta-logical style properties without actually needing a meta-logic [5].

The core UTP types include predicates $[\alpha]upred$, and (homogeneous) relations $[\alpha]hrel$. Operators are denoted using lenses and lifted HOL functions. An important operator is substitution, $\sigma \dagger P$, which applies a state update function $\sigma :: \alpha \to \alpha$ to an expression, and replaces variables in a similar way to syntactic substitution. Substitutions can be built using lens updates $\sigma(x \mapsto v)$, for $x :: \tau \Longrightarrow \alpha$ and $v :: [\tau, \alpha]uexpr$, and we use the notation $(\!|x_1 \mapsto v_1, \cdots, x_n \mapsto v_n|\!)$ for a substitution in n variables. Substitution theorems can be applied with the simplifier to perform symbolic evaluation of an expression.

All the theorems and results that we present in this paper have been mechanically validated in Isabelle/UTP, and the proofs can be found in our repository[3].

2.4 Stateful-Failure Reactive Designs

RoboChart is a reactive language, where controllers exchange events with one another and the robotic platform or the environment. Reactive programs can make decisions both internally, based on the evaluation of their own state, and externally, by offering several events. Consequently, they pause at particular quiescent points during execution, when awaiting a communication. Unlike sequential programs, they need not terminate but may run indefinitely.

The UTP theory of stateful-failure reactive designs [14, 15] exists to give denotational semantics to reactive programming languages, such as CSP [2], *Circus* [3], and rCOS [16]. It is a relational version of the stable failures-divergences semantic model, as originally defined in the UTP book [2,17] using events traces and refusal sets, but extended with state variables. Its healthiness condition, ***NCSP***, which we previously mechanised [14], characterises relations that extend the trace, update variables, and refuse certain events in quiescent phases. The signature includes unbounded nondeterministic choice ($\bigsqcap_{i \in I} P(i)$), conditional ($P \lhd b \rhd Q$), and sequential composition ($P \fatsemi Q$). $[\![\textbf{\textit{NCSP}}]\!]$ forms a complete lattice under $\sqsubseteq$, with top element ***miracle*** and bottom ***chaos***, and also a Kleene algebra [14], which allows reasoning about iterative reactive programs.

The signature also contains several specialised operators. Event action, ***do***(e), describes the execution of an event expression e, that ranges over state variables. When activated, it waits for e to occur, and then it terminates. Generalised assignment ($\langle \sigma \rangle_a$) uses a substitution σ to update the state, following Back [18]. Basic assignment can be defined as $(x := v) \triangleq \langle (\!|x \mapsto v|\!) \rangle_a$, and a unit as ***skip*** $\triangleq \langle id \rangle_a$. External choice, $\Box\, i \in A \bullet P(i)$ indexed by set A, as in CSP, permits one of the branches to resolve either by an event, or by termination. A binary choice $P \,\Box\, Q$ is denoted by $\Box\, X \in \{P, Q\} \bullet X$. A guard, $b \,\&\, P$, executes P when b is true, and is otherwise equivalent to ***stop***, the deadlocked action. These operators obey several algebraic laws [14]; a small selection is below.

[3] https://github.com/isabelle-utp/utp-main/tree/master/robochart/untimed

Theorem 2.2. *If P is **NCSP**-healthy, then the following identities hold:*

$$\textbf{\textit{miracle}} \fatsemi P = \textbf{\textit{miracle}} \tag{1}$$

$$\langle \sigma \rangle_a \fatsemi P = \sigma \dagger P \tag{2}$$

$$(\textbf{\textit{do}}(a) \Box \textbf{\textit{do}}(b)) \fatsemi P = \textbf{\textit{do}}(a) \fatsemi P \Box \textbf{\textit{do}}(b) \fatsemi P \tag{3}$$

(1) states that miracle is a left annihilator for sequential composition. (2) allows us to push an assignment into a successor program by inserting a substitution. (3) allows us to right distribute through an external choice of events.

Our theory supports specifications using reactive contracts: $[P \vdash Q \mid R]$. It consists of three predicates over the state variables, trace variable tt, and refusal set variable ref. P characterises assumptions of the initial state and trace, Q characterises quiescent behaviours, and R characterises terminating behaviours. Our previous result [14] shows that any reactive program can be denoted using a reactive contract, which can be calculated by equational laws. This enables a verification strategy that checks refinements between a specification and implementation contract, and has been implemented in a tactic called rdes-refine [19], that can be used to check for deadlock [14], and which we employ in this paper.

3 Foundations for State Diagrams

In this section we extend the theory of reactive designs with constructs necessary to denote state machines, and prove several theorems, notably an induction law for iterative programs. Although these programming constructs are rather standard, we consider their semantics in the reactive programming paradigm, rather than in the standard sequential programming setting. It is a pleasing aspect of our approach that standard laws hold in this much richer context.

State machines describe how to transition from one node to another. The main construct we use to denote them is a reactive version of Dijkstra's guarded iteration statement [20] $\textbf{\textit{do}}\ i \in I \bullet b(i) \rightarrowtriangle P(i)\ \textbf{\textit{od}}$, which repeatedly selects an indexed statement $P(i)$, based on whether its respective guard $b(i)$ is true. I is an index set, which when finite gives rise to the more programmatic form of $\textbf{\textit{do}}\ b_1 \rightarrowtriangle P_1 \mid \cdots \mid b_n \rightarrowtriangle P_n\ \textbf{\textit{od}}$. We begin by defining Dijkstra's alternation construct, $\textbf{\textit{if}}\ i \in I \bullet b(i) \rightarrowtriangle P(i)\ \textbf{\textit{fi}}$ [20], which underlies iteration.

Definition 3.1 (Guarded Commands, Assumptions, and Alternation).

$$b \rightarrowtriangle P \triangleq P \lhd b \rhd \textbf{\textit{miracle}}$$

$$[b] \triangleq b \rightarrowtriangle \textbf{\textit{skip}}$$

$$\textbf{\textit{if}}\ i \in I \bullet b(i) \rightarrowtriangle P(i)\ \textbf{\textit{fi}} \triangleq \left(\textstyle\bigsqcap_{i \in I}\ b(i) \rightarrowtriangle P(i)\right) \sqcap \left(\left(\neg \textstyle\bigvee_{i \in I}\ b(i)\right) \rightarrowtriangle \textbf{\textit{chaos}}\right)$$

$b \rightarrowtriangle P$ is a "naked" guarded command [21]. Its behaviour is P when b is true, and miraculous otherwise, meaning it is impossible to execute. By Theorem 2.2, ***miracle*** is a left annihilator for sequential composition, and so any following behaviour is excluded when b is false. An assumption $[b]$ guards ***skip*** with b,

and thus holds all variables constant when b is true, and is otherwise miraculous. These operators are both closed under $[\![\textbf{\textit{NCSP}}]\!]$ since they are defined only in terms of healthy elements $\lhd \cdot \rhd$, ***miracle***, and ***skip***.

Alternation is a non-deterministic choice of guarded commands. When $b(i)$ is true for $i \in I$, $P(i)$ can be executed. Any command which has $b(i)$ false evaluates to ***miracle*** and thus is eliminated. If no $b(i)$ is true, then its behaviour is ***chaos***. If multiple $b(i)$ are true then one of the corresponding $P(i)$ is nondeterministically selected. Alternation is closed under $[\![\textbf{\textit{NCSP}}]\!]$ since it comprises only healthy elements. From this definition we can prove a number of characteristic laws.

Theorem 3.2. *If,* $\forall\, i \bullet P(i)$ *is* ***NCSP****, then the following identities hold:*

$$\textbf{\textit{if}}\ i \in \emptyset \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{fi}} = \textbf{\textit{chaos}} \tag{1}$$

$$\textbf{\textit{if}}\ i \in \{k\} \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{fi}} = P(k) \lhd b(k) \rhd \textbf{\textit{chaos}} \tag{2}$$

$$\left[\bigvee_{i \in I}\ b(i)\right] ;\ \textbf{\textit{if}}\ i \in I \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{fi}} = \left(\bigsqcap_{i \in I}\ b(i) \twoheadrightarrow P(i)\right) \tag{3}$$

In words, (1) shows that alternation over an empty set presents no options, and so is equivalent to ***chaos***; (2) shows that a singleton alternation can be rewritten as a binary conditional; (3) shows that, if we assume that one of its branches is true, then an alternation degenerates to a nondeterministic choice.

We now define guarded iteration as the iteration of the corresponding alternation whilst at least one of the guards remains true.

Definition 3.3 (Guarded Iteration).

$$\textbf{\textit{do}}\ i \in I \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{od}} \triangleq \left(\bigvee_{i \in I}\ b(i)\right) \circledast (\textbf{\textit{if}}\ i \in I \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{fi}})$$

We use the reactive while loop $(b \circledast P)$ to encode the operator, and can thus utilise our previous results [14] to reason about it. In keeping with the reactive programming paradigm, this while loop can pause during execution to await interaction, and it also need not terminate. However, in order to ensure that the underlying fixed point exists, we assume that for all $i \in I$, $P(i)$ is productive [10]: that is, it produces at least one event whenever it terminates. This ensures that divergence caused by an infinite loop is avoided. Iteration is closed under $[\![\textbf{\textit{NCSP}}]\!]$, since the while loop and alternation both are.

We can now prove the following fundamental refinement law for iteration.

Theorem 3.4 (Iteration Induction). *If,* $\forall\, i \bullet P(i)$ *is* ***NCSP****, then:*

$$\frac{\begin{array}{ll} \forall\, i \in A \bullet P(i)\ \textit{is Productive} & S \sqsubseteq I ;\ [\bigwedge_{i \in A} (\neg b(i))] \\ \forall\, i \in A \bullet S \sqsubseteq I ;\ [b(i)] ;\ P(i) & \forall\, i \in A \bullet S \sqsubseteq S ;\ [b(i)] ;\ P(i) \end{array}}{S \sqsubseteq I ;\ \textbf{\textit{do}}\ i \in A \bullet b(i) \twoheadrightarrow P(i)\ \textbf{\textit{od}}}$$

The law states the provisos under which an iteration, with initialiser I, preserves invariant S. These are: (1) every branch is productive; (2) if I causes the iteration to exit immediately then S is satisfied; (3) for any $i \in A$ if I holds initially, $b(i)$ is true, and $P(i)$ executes, then S is satisfied (base case); and (4) for any $i \in A$ if $b(i)$ is true, and $P(i)$ executes, then S is satisfied (inductive case). This law forms the basis for our verification strategy.

4 Mechanised Reactive Programs

In this section we turn our reactive design theory into an Isabelle/HOL type, so that we can use the type system to ensure well-formedness of reactive programs, which supports our verification strategy. The type allows efficient proof and use of the simplifier to perform rewriting and also symbolic evaluation so that assignments can be pushed forward and substitutions applied. We use it both to encode state machine actions in §5, and the dynamic semantics in §6. We first describe a general result for mechanising programs, apply it to reactive programs, and also introduce a novel operator to express frame extension.

In UTP, all programs are unified by enconding them in the alphabetised relational calculus. Programs in different languages of various paradigms therefore have a common mathematical form, and can be both compared and semantically integrated. This idea is retained in Isabelle/UTP by having all programs occupy the type $[\alpha]hrel$, with a suitably specialised alphabet type α [12].

In Isabelle/UTP, we characterise a theory by (1) an alphabet type $\mathcal{T}$, that may be parametric; and (2) a healthiness function, $\mathbf{H} :: [\mathcal{T}]hrel \to [\mathcal{T}]hrel$. The signature of the theory is a collection of constructors with k arguments of the form $f_i :: ([\mathcal{T}]hrel)^k \to [\mathcal{T}]hrel$, each of which has a closure theorem

$$f\text{-}H\text{-}closed : [\![P_1 \ is \ \mathbf{H};\ \cdots\ ;\ P_k \ is \ \mathbf{H}]\!] \Longrightarrow f(P_1, \cdots, P_k) \ is \ \mathbf{H}$$

that ensures the operator constructs healthy elements, provided its parameters are all healthy. A theory also typically has a set of algebraic laws of programming, like those in Theorem 2.2, that can be applied to reasoning about elements of the theory and thence to produce verification tools [14,19].

This approach has several advantages for theory engineering. There is a unified notion of refinement that can be applied across semantic domains. Operators like nondeterminsitic choice ($\sqcap$) and sequential composition ($\fatsemi$) can occupy several theories, which facilitates generality and semantic integration. The UTP approach means that theories can be both combined and specialised.

However, there is a practical downside, which is that the programming theorems, such as those in Theorem 2.2, are subject to healthiness of the constituent parameters, and therefore it is necessary to first invoke the closure theorems. In the context of verification, constantly proving closure can be very inefficient, particularly for larger programs. This is because Isabelle's simplifier works best when invoked with pure equations $f(x_1, \cdots, x_n) \equiv y$ with minimal provisos.

Our solution uses the Isabelle type system to shoulder the burden of closure proof. We use the ***typedef*** mechanism, which creates a new type T from a non-empty subset $A :: \mathbb{P}(U)$ of existing type U. For a UTP theory, we create a type with $A = [\![\mathbf{H}]\!]$, which is a subset of the UTP relations.

In order to obtain the signature for the new type, we utilise the lifting package [22], whose objective is to define operators on T in terms of operators on U, provided that each operator is closed under A. Specifically, if f is a signature operator in k arguments, then we can create a lifted operator $\widehat{f} :: T^k \to T$ using Isabelle's ***lift-definition*** command [22]. This raises a proof obligation that

$f \in [\![\boldsymbol{H}]\!]^k \to [\![\boldsymbol{H}]\!]$, which can be discharged by the corresponding closure theorem. Programs constructed from the lifted operators are well-formed by construction.

Finally, the lifting package provides a tactic *transfer* that helps to prove theorems on lifted operators through corresponding theorems on the unlifted ones. Specifically, we can prove a theorem like $\widehat{f}(P_1, \cdots, P_k) = \widehat{g}(P_1, \cdots, P_k)$, where $P_i :: T$ is a free variable, by converting it to a theorem of the form

$$[\![Q_1 \ \textit{is}\ \boldsymbol{H};\ \cdots;\ Q_k \ \textit{is}\ \boldsymbol{H}]\!] \Longrightarrow f(Q_1, \cdots, Q_k) = g(Q_1, \cdots, Q_k).$$

This means the closure properties of each parameter Q_i can be utilised in discharging provisos of the corresponding UTP theorems, but the lifted theorems do not require them. We will now use this technique for our reactive program type.

The reactive designs alphabet is $[s, e]$*st-csp*, for state space s and event type e. Function ***NCSP*** [14], of type $[[s, e]\textit{st-csp}]\textit{hrel} \to [[s, e]\textit{st-csp}]\textit{hrel}$, characterises the theory we use it to define the reactive program type, $[s, e]$*Action*. We then lift each of the theory operators described in §2 and §3. For example, guard is a function $(b \mathbin{\&} P) :: [s, e]\textit{Action}$, for $b :: [s]\textit{upred}$ and $P :: [s, e]\textit{Action}$. For the action language, we encode functions for event synchronisation $e \triangleq \textbf{\textit{do}}(e)$, send $e!v \triangleq \textbf{\textit{do}}(e.v)$, and receive $e?x \triangleq \Box\, v \bullet \textbf{\textit{do}}(e.v) \fatsemi x := v$. From these lifted definitions, and utilising the *transfer* tactic, all of the laws in Theorems 2.2 and 3.4 can be recast for the new operators, but without the closure conditions. Then, we can prove substitution laws for $\sigma \dagger P$, for $\sigma :: s \to s$ and $P :: [s, e]\textit{Action}$.

Theorem 4.1 (Symbolic Evaluation Laws).

$$\begin{aligned}
\sigma \dagger [b] &= [\sigma \dagger b] \fatsemi \langle \sigma \rangle_a & \sigma \dagger (P \mathbin{\Box} Q) &= (\sigma \dagger P) \mathbin{\Box} (\sigma \dagger Q) \\
\sigma \dagger (P \fatsemi Q) &= (\sigma \dagger P) \fatsemi Q & \sigma \dagger (b \mathbin{\&} P) &= (\sigma \dagger b) \mathbin{\&} (\sigma \dagger P) \\
\sigma \dagger \langle \rho \rangle_a &= \langle \rho \circ \sigma \rangle_a & \sigma \dagger e!v &= e!(\sigma \dagger v) \fatsemi \langle \sigma \rangle_a
\end{aligned}$$

These laws show how substitution applies and distributes through the operators. In combination with the assignment law of Theorem 2.2(2), they can be used to apply state updates. For example, one can automatically prove that

$$(x := 2 \fatsemi y := (3 * x) \fatsemi e!(x + y)) \;=\; (e!8 \fatsemi \langle x \mapsto 2, y \mapsto 6 \rangle_a)$$

since we can combine the assignments and push them through the send event.

To denote state machines, we need a special variable (*actv*) to record the currently active node. This is semantic machinery, and no action is permitted access to it. We impose this constraint via frame extension: $a{:}[P]^+ :: [s_1, e]\textit{Action}$, for $a :: s_2 \Longrightarrow s_1$ and $P :: [s_2, e]\textit{Action}$, that extends the alphabet of P. It is similar to a frame in refinement calculus [21], which prevents modification of variables, but also uses the type system to statically prevent access to them. Lens a identifies a subregion α of the larger alphabet β, that P acts upon. Intuitively, α is the set of state machine variables, and β this set extended with *actv*. P can only modify variables within α, and others are held constant. We prove laws for this operator, which will also be used in calculating the semantics.

Theorem 4.2 (Frame Extension Laws).

$$a{:}[P \fatsemi Q]^+ = a{:}[P]^+ \fatsemi a{:}[Q]^+ \qquad a{:}[e?x]^+ = e?(a{:}x) \qquad a{:}[x := v]^+ = a{:}x := v$$

Frame extension distributes through sequential composition. For operators like event receive and assignment, the variable is extended by the lens a, which can considered as a namespace operator. This then completes the mechanised reactive language. In the next section we will mechanise the static semantics.

5 Static Semantics

In this section we formalise a state machine meta-model in Isabelle/HOL, which describes the variables, transitions, and nodes. The meta-model, presented below, is based on the untimed subset of RoboChart, but note that our use of UTP ensures that our work is extensible to more advanced semantic domains [7,10,11]. For now we omit constructs concerned with interfaces, operations, shared variables, during actions, and hierarchy, and focus on basic machines.

Definition 5.1 (State Machine Meta-Model).

$$\begin{aligned} \mathit{StMach} &:= \textbf{statemachine}\ \mathit{ID} \\ &\quad \textbf{vars}\ \mathit{NameDecl}^*\ \textbf{events}\ \mathit{NameDecl}^*\ \textbf{states}\ \mathit{NodeDecl}^* \\ &\quad \textbf{initial}\ \mathit{ID}\ \textbf{finals}\ \mathit{ID}^*\ \textbf{transitions}\ \mathit{TransDecl}^* \\ \mathit{NameDecl} &:= \mathit{ID}\ [\textbf{:}\ \mathit{Type}] \\ \mathit{NodeDecl} &:= \mathit{ID}\ \textbf{entry}\ \mathit{Action}\ \textbf{exit}\ \mathit{Action} \\ \mathit{TransDecl} &:= \mathit{ID}\ \textbf{from}\ \mathit{ID}\ \textbf{to}\ \mathit{ID}\ \textbf{trigger}\ \mathit{Event}\ \textbf{condition}\ \mathit{Expr}\ \textbf{action}\ \mathit{Action} \end{aligned}$$

A state machine is composed of an identifier, variable declarations, event declarations, state declarations, an initial state identifier, final state identifiers, and transition declarations. Each variable and event consists of a name and a type. A state declaration consists of an identifier, entry action, and exit action. A transition declaration consists of an identifier, two state identifiers for the source and target nodes, a trigger event, a condition, and a body action. Whilst we do not directly consider hierarchy, this can be treated by flattening out substates.

We implement the meta-model syntax using Isabelle's parser, and implement record types $[s, e]\mathit{Node}$ and $[s, e]\mathit{Transition}$, that correspond to the *NodeDecl* and *TransDecl* syntactic categories. They are both parametric over the state-space s and event types e. *Node* has fields $\mathit{nname} :: \mathit{string}$, $\mathit{nentry} :: [s, e]\mathit{Action}$, and $\mathit{nexit} :: [s, e]\mathit{Action}$, that contain the name, entry action, and exit action. *Transition* has fields $\mathit{src} :: \mathit{string}$, $\mathit{tgt} :: \mathit{string}$, $\mathit{trig} :: [s, e]\mathit{Action}$, $\mathit{cond} :: [s]\mathit{upred}$, and $\mathit{act} :: [s, e]\mathit{Action}$, that contain the source and target, the trigger, the condition, and the body. We then create a record type to represent the state machine.

Definition 5.2 (State Machine Record Type).

$$\textbf{record}\,[s, e]\mathit{StMach} = \begin{array}{ll} \mathit{init} :: \mathit{ID} & \mathit{finals} :: [\mathit{ID}]\mathit{list} \\ \mathit{nodes} :: [[s, e]\mathit{Node}]\mathit{list} & \mathit{transs} :: [[s, e]\mathit{Transition}]\mathit{list} \end{array}$$

It declares four fields for the initial state identifier (*init*), final states identifiers (*finals*), nodes definitions (*nodes*), and transition definitions (*transs*), and constitutes the static semantics. Since this corresponds to the meta-model, and to

ensure a direct correspondence with the parser, we do not directly use sets and maps, but only lists in our structure. We will later derive views onto the data structure above, that build on well-formedness constraints.

Below, we show how syntactic machines are translated to Isabelle definitions.

Definition 5.3 (Static Semantics Translation).

$$
\begin{array}{l}
\textbf{\textit{statemachine}}\ s \\
\quad \textbf{\textit{vars}}\ x_1 : \tau_1^v\ \cdots\ x_i : \tau_i^v \\
\quad \textbf{\textit{events}}\ e_1 : \tau_1^e\ \cdots\ e_j\ : \tau_j^e \\
\quad \textbf{\textit{states}}\ s_1 \cdots s_k\ \textbf{\textit{initial}}\ ini \\
\quad \textbf{\textit{finals}}\ f_1 \cdots f_m \\
\quad \textbf{\textit{transitions}}\ t_1 \cdots t_n
\end{array}
\implies
\begin{array}{l}
\textbf{\textit{alphabet}}\ s\text{-}alpha = x_1 : \tau_1^v\ \cdots\ x_i : \tau_i^v \\
\textbf{\textit{datatype}}\ s\text{-}ev = \boldsymbol{\epsilon} \mid e_1\ t_1^e \mid \cdots \mid e_j\ t_j^e \\
\textbf{\textit{definition}}\ machine :: [s\text{-}alpha, s\text{-}ev]\, StMach \\
\textit{where}\ machine = \begin{array}{l} (\!| init = ini, \\ \ finals = [f_1 \cdots f_m], \\ \ states = [s_1 \cdots s_k], \\ \ transs = [t_1 \cdots t_n] |\!) \end{array} \\
\textbf{\textit{definition}}\ semantics = [\![machine]\!]_{\mathsf{M}}
\end{array}
$$

For each machine, a new alphabet is created, which gives rise to a HOL record type *s-alpha*, and lenses for each field of the form $t_i^v \implies s\text{-}alph$. For the events, an algebraic datatype *s-ev* is created with constructors corresponding to each of them. We create a distinguished event $\boldsymbol{\epsilon}$ that will be used in transitions with explicit trigger and ensures productivity. The overall machine static semantics is then contained in *machine*. We also define *semantics* that contains the dynamic semantics in terms of the semantic function $[\![\cdot]\!]_{\mathsf{M}}$ that we describe in §6.

Elements of the meta-model are potentially not well-formed, for example specifying an initial state without a corresponding state declaration, and therefore it is necessary to formalise well-formedness. RoboTool enforces a number of well-formedness constraints [23], and we here formalise the subset needed to ensure the dynamic semantics given in §6 can be generated. We need some derived functions for this, and so we define $nnames \triangleq set(map\ nname\ (nodes))$, which calculates the set of node names, and *fnames*, which calculates the set of final node names. We can now specify our well-formedness constraints.

Definition 5.4. *A state machine is well-formed if it satisfies these constraints:*

1. The list of node identifiers forms a set: $distinct(map\ nname\ (nodes))$
2. The initial identifier is defined: $init \in nnames$
3. The initial identifier is not final: $init \notin finals$
4. Every transition's source node is defined and non-final: $\forall\, t \in transs \bullet src(t) \in nnames \setminus fnames$
5. Every transition's target node is defined: $\forall\, t \in transs \bullet tgt(t) \in nnames$

We have implemented them in Isabelle/HOL, along with a proof tactic called *check-machine* that discharges them automatically when a generated static semantics is well-formed, and ensure that crucial theorems are available to the dynamic semantics. In practice, any machine accepted by RoboTool is well-formed, and so this tactic simply provides a proof of that fact to Isabelle/HOL.

```
statemachine GasAnalysis [
  vars sts::Status  gs::"GasSensor list"  ins::Intensity  anl::real
  events resume  stop  turn::real  gas::"GasSensor list"
  states InitState NoGas Reading FinalState
         Analysis: "entry sts := «analysis»(&gs)ₐ"
         GasDetected: "entry ins := «intensity»(&gs)ₐ"
  initial InitState finals FinalState
  transitions
    t1: "from InitState to NoGas action gs := ⟨⟩; anl := 0"
    t2: "from NoGas to Analysis trigger gas?(gs)"
    t3: "from Analysis to NoGas condition &sts =ᵤ «noGas» action resume"
    t4: "from Analysis to GasDetected condition (&sts =ᵤ «gasD»)"
    t5: "from GasDetected to FinalState condition «goreq»(&ins,«thr»)ₐ action stop"
    t6: "from GasDetected to Reading condition ¬ «goreq»(&ins,«thr»)ₐ
         action anl := «location»(&gs)ₐ ; turn!(&anl)"
    t7: "from Reading to Analysis trigger gas?(gs)" ]
```

Fig. 2. State machine notation in Isabelle/UTP

In a well-formed machine every node has a unique identifier. Therefore, using Definition 5.4, we construct two finite partial functions, $\mathit{nmap} :: \mathit{ID} \nrightarrow [s, e]\mathit{Node}$ and $\mathit{tmap} :: \mathit{ID} \nrightarrow [s, e]\mathit{Transition\,list}$, that obtain the node definition and list of transitions associated with a particular node identifier, respectively, whose domains are both equal to *nnames*. We also define $\mathit{ninit} \triangleq \mathit{nmap}\,\mathit{init}$, to be the definition of the initial node, and *inters* to be the set of nodes that are not final. Using well-formedness we can then prove the following theorems.

Theorem 5.5 (Well-formedness Properties).

1. All nodes are identified: $\forall\, n \in \mathit{set}(\mathit{nodes}) \bullet \mathit{nmap}(\mathit{nname}(n)) = n$
2. The initial node is defined: $\mathit{ninit} \in \mathit{set}(\mathit{nodes})$
3. The name of the initial node is correct: $\mathit{nname}(\mathit{ninit}) = \mathit{init}$

These theorems allow us to extract the unique node for each identifier, and in particular for the initial node. Thus Isabelle/HOL can parse a state machine definition, construct a static semantics for it, and ensure this semantics type checks and is well-formed. The resulting Isabelle command is illustrated in Figure 2 that encodes the GasAnalysis state machine of Figure 1.

6 Dynamic Semantics

In this section we describe the behaviour of a state machine using the reactive program domain we mechanised in §4. The RoboChart reference semantics [23] represents a state machine as a parallel composition of CSP processes that represent the individual variables and states. Variable access and state orchestration is modelled by communications between them. Here, we capture a simpler sequentialised semantics using guarded iteration, which eases verification. In particular, state variables have a direct semantics, and require no communication. The relation between these two semantics can be formalised by an automated refinement strategy that reduces parallel to sequential composition [3].

We first define alphabet type $[s]\mathit{rcst}$, parametrised by the state space type s, and consisting of lenses $\mathit{actv} :: \mathit{ID} \Longrightarrow [s]\mathit{rcst}$ and $\mathbf{r} :: s \Longrightarrow [s]\mathit{rcst}$, the former of

which record the currently active state, and the latter of which projects the state machine variable space. No action is permitted to refer to *actv*, a constraint that we impose through the frame extension $\boldsymbol{r}{:}[P]^+$.

We describe the dynamic semantics of a state diagram using three functions.

Definition 6.1 (Dynamic Semantics).

$$[\![M]\!]_{\mathsf{M}} \triangleq \left(\begin{array}{l} actv := init_M \fatsemi \\ \textbf{\textit{do}}\ N \in set(inters_M) \bullet actv = nname(N) \rightarrowtriangle M \models [\![N]\!]_{\mathsf{N}}\ \textbf{\textit{od}} \end{array}\right)$$

$$M \models [\![N]\!]_{\mathsf{N}} \triangleq \boldsymbol{r}{:}[nentry(N)]^+ \fatsemi \left(\Box\, t \in \mathit{tmap}_M(nname(N)) \bullet M, N \models [\![t]\!]_{\mathsf{T}}\right)$$

$$M, N \models [\![t]\!]_{\mathsf{T}} \triangleq \boldsymbol{r}{:}[cond(t)\ \&\ trig(t) \fatsemi nexit(N) \fatsemi action(t)]^+ \fatsemi actv := tgt(t)$$

The function $[\![\cdot]\!]_{\mathsf{M}} :: [s, e]StMach \rightarrow [[s]rcst, e]Action$ calculates the overall behavioural semantics. It first sets *actv* to the initial node identifier, and then enters a do iteration indexed by all non-final nodes. If a final node is selected, then the iteration terminates. In each iteration, the node N that is named by *actv* is selected, and the semantics for it is calculated using $M \models [\![N]\!]_{\mathsf{N}}$.

When in a node, the entry action is first executed using *nentry*, and then an external choice is presented over all transitions associated with N, which are calculated using *tmap*. The entry and exit actions do not have *actv* in their alphabet, and therefore we apply frame extensions to them. The semantics of a transition, $M, N \models [\![t]\!]_{\mathsf{T}}$, is guarded by the transition condition, and awaits the trigger event. Once this occurs, the exit action of N is executed, followed by the transition action, and finally *actv* is updated with the target node identifier.

The output of the semantics is an iterative program with one branch for every non-final state. To illustrate, we below generate the denotational semantics for the GasAnalysis state machine given in Figure 1.

Example 6.2 (GasAnalysis Dynamic Semantics).

$$\begin{array}{l} actv := InitState \fatsemi \\ \textbf{\textit{do}} \\ \quad actv = InitState \rightarrow \boldsymbol{\epsilon} \fatsemi \boldsymbol{r}{:}gs := \langle\rangle \fatsemi \boldsymbol{r}{:}anl := 0 \fatsemi actv := NoGas \\ \mid actv = NoGas \rightarrow gas?\boldsymbol{r}{:}gs \fatsemi actv := Analysis \\ \mid actv = Analysis \rightarrow \\ \quad \boldsymbol{r}{:}sts := analysis(\boldsymbol{r}{:}gs) \fatsemi \left(\begin{array}{l} \boldsymbol{r}{:}sts = noGas\ \&\ \boldsymbol{\epsilon} \fatsemi resume \fatsemi actv := NoGas \\ \Box\ \boldsymbol{r}{:}sts = gasD\ \&\ \boldsymbol{\epsilon} \fatsemi actv := GasDetected \end{array}\right) \\ \mid actv = GasDetected \rightarrow \boldsymbol{r}{:}ins := intensity(\boldsymbol{r}{:}gs) \fatsemi \\ \qquad \left(\begin{array}{l} goreq(ins, thr)\ \&\ \boldsymbol{\epsilon} \fatsemi stop \fatsemi actv := FinalState \\ \Box\ (\neg goreq(ins, thr))\ \&\ \boldsymbol{\epsilon} \fatsemi \begin{array}{l} \boldsymbol{r}{:}anl := location(\boldsymbol{r}{:}gs) \fatsemi \\ turn!(\boldsymbol{r}{:}anl) \fatsemi actv := Reading \end{array} \end{array}\right) \\ \mid actv = Reading \rightarrow gas?\boldsymbol{r}{:}gs \fatsemi actv := Analysis \\ \textbf{\textit{od}} \end{array}$$

In order to yield a more concise definition, we have also applied the action simplification laws given in §4. In particular, the frame extensions have all been expanded so that the state variables are explicitly qualified by lens $\boldsymbol{r}$.

In order to verify such state machines, we need a specialised refinement introduction law. Using our well-formedness theorem, we can specialise Theorem 3.4.

Theorem 6.3. *The semantics of a state machine M refines a reactive invariant specification S, that is $S \sqsubseteq [\![M]\!]_{\mathsf{M}}$, provided that the following conditions hold:*

1. *M is well-formed according to Definition 5.4;*
2. *the initial node establishes the invariant — $S \sqsubseteq M \models [\![ninit_M]\!]_{\mathsf{N}}$;*
3. *every non-final node preserves S — $\forall\, N \in inters_M \bullet S \sqsubseteq S \fatsemi (M \models [\![N]\!]_{\mathsf{N}})$.*

Proof. By application of Theorem 3.4, and utilising trigger productivity. □

We now have all the infrastructure needed for verification of state machines, and in the next section we describe our verification strategy and tool.

7 Verification Approach

In this section we will use the collected results presented in the previous sections to define a verification strategy for state machines, and exemplify its use in verifying deadlock freedom. Our approach utilises Theorem 6.3 and our contractual refinement tactic, rdes-refine, to prove that every state of a state machine satisfies a given invariant, which is specified as a reactive contract. The overall workflow for description and verification of a state machine is given by the following steps:

1. parse, type check, and compile the state machine definition;
2. check well-formedness (Definition 5.4) using the *check-machine* tactic;
3. calculate denotational semantics, resulting in a reactive program;
4. perform algebraic simplification and symbolic evaluation (Thms 2.2, 4.1);
5. apply Theorem 6.3 to produce sequential refinement proof obligations;
6. apply rdes-refine to each goal, which may result in residual proof obligations;
7. attempt to discharge each remaining proof obligation using sledgehammer [13].

Diagrammatic editors, like RoboTool, can be integrated with this by implementing a serialiser for the underlying meta-model. The workflow can be completely automated since there is no need to enter manual proofs, and the final proof obligations are discharged by automated theorem provers. If proof fails, Isabelle/HOL has the nitpick [13] counterexample generator that can be used for debugging. This means that the workflow can be hidden behind a graphical tool.

We can use the verification procedure to check deadlock freedom of a state machine using the reactive contract $\mathit{dlockf} \triangleq [\, \mathbf{true} \vdash \exists\, e \bullet e \notin \mathsf{ref} \mid \mathbf{true} \,]$, an invariant specification which states that in all quiescent observations, there is always an event that this not being refused. In other words, at least one event is always enabled; this is the meaning of deadlock freedom. We can use this contract to check the GasAnalysis state machine. For a sequential machine, deadlock freedom means that it is not possible to enter a state and then make no further progress. Such a situation can occur if the outgoing transitions can all be disabled simultaneously if, for example, their guards do not cover all possibilities.

```
goal (6 subgoals):
 1. dlockf ⊑ ε ; ⟨[&r:gs ↦s ⟨⟩, &r:anl ↦s 0, &rc_ctrl ↦s «''NoGas''»]⟩a
 2. dlockf ⊑
    ((«analysis»(&r:gs)a =u «gasD») &
     ε ; (r:sts), rc_ctrl := «analysis»(&r:gs)a, «''GasDetected''») □
    ((«analysis»(&r:gs)a =u «noGas») &
     ε ; resume ; (r:sts), rc_ctrl := «analysis»(&r:gs)a, «''NoGas''»)
 3. dlockf ⊑
    ((¬ «goreq»((«intensity»(&r:gs)a, «thr»)u)a) &
     ε ; turn!(«location»(&r:gs)a) ;
          ⟨[&r:ins ↦s «intensity»(&r:gs)a, &r:anl ↦s «location»(&r:gs)a, &rc_ctrl ↦s
            «''Reading''»]⟩a) □
    («goreq»((«intensity»(&r:gs)a, «thr»)u)a &
     ε ; stop ; (r:ins), rc_ctrl := «intensity»(&r:gs)a, «''FinalState''»)
```

Fig. 3. Selection of deadlock freedom proof obligations in Isabelle/UTP

The result of applying the verification procedure up to step 5 is shown in Figure 3. At this stage, the semantics for each node has been generated, and deadlock freedom refinement conjectures need to be proved. Isabelle generates 6 subgoals, 3 of which are shown, since it is necessary to demonstrate that the invariant is satisfied by the initial state and each non-final state. The first goal corresponds to the initial state, where no event occurs and the variables *gs* and *anl*, along with *actv*, are all assigned. The second goal corresponds to the Analysis state. The state body has been further simplified from the form shown in Figure 6.2, since symbolic evaluation has pushed the entry action through the transition external choice, and into the two guards. This is also the case for the third goal, which corresponds to the more complex GasDetected state.

The penultimate step involves execution of the rdes-refine tactic on each of the subgoals. This results in three subgoals for each goal, a total of 18 proof obligations, all of which are first order predicates. The majority can be discharged using the relational calculus tactic rel-auto. However, in this case three remaining goals remain that are HOL predicates. One of these goals is related to the Analysis state, and requires that the constructors *noGas* and *gasD* of *Status* are the only possibilities for *sts*. If there was a third possibility, there would be a deadlock and so it is necessary to check this. We execute sledgehammer on each of the three goals, which provides proofs and so completes the deadlock freedom check.

8 Conclusions and Related Work

In this paper we presented a prototype verification strategy for state-machine diagrams in Isabelle/UTP by utilising the theory of stateful-failure reactive designs, and automated proof facilities. We extended our UTP theory with the guarded iteration construct, which is the foundation of sequential state machines, and proved a crucial induction law, and adapted it to an efficient implementation of reactive programs. We created a static semantics of state machines in Isabelle/HOL, including well-formedness checks, and a dynamic semantics that generates a reactive program. Finally, we used this to describe a verification approach that utilises reactive contract refinement and iterative induction.

In future work, we will expand our semantics to handle additional features of RoboChart. Hierarchy, could be handled by having the *actv* variable hold a list of nodes, and during actions by implementing a reactive design interruption operator [24]. Moreover, we are developing reasoning facilities for parallel composition, hiding, and renaming for expression of concurrent state machines, which extends our existing work [14, 19]. This will greatly increase verification capabilities for robotic and component-based systems, and allow us to handle asynchronous communication and shared variables. A challenge here is handling assumptions and guarantees between parallel components, but we believe that abstraction of a state machines to invariants, using our results, could make this tractable. We will also explore other reasoning approaches, such as use of the simplifier to algebraically transform state machines to equivalent forms.

Going further, we emphasise that our UTP theory hierarchy supports more advanced semantic paradigms. We will therefore develop a mechanised theory of timed reactive designs, based on existing work [7, 10], and use this to denote the timing constructs of RoboChart state machines. We are also developing a UTP theory of probability [11], and will use it to handle probabilistic junctions. Finally, we also have a theory of hybrid reactive designs [9, 10], that we believe could be used to support notations like hybrid state machines in this setting.

In related work, while a number of state machine notations exist, such as UML and Stateflow, to the best of our knowledge, they provide limited support for formal verification. While formalisations of such languages have been proposed [25, 26], they traditionally address a subset of the target notation or focus on model checking. Other approaches such as [27], similarly restrict themselves to model checking or other forms of automatic verification, which have strong limitations on both the types of systems that can be analysed (mostly finite) and the kinds of properties that can be checked (schedulability, temporal logic, etc). We differ in that our approach is extensible, fully automated, and can handle infinite state system with non-trivial types. Also, our verification laws have been mechanically validated with respect only to the axioms of Isabelle/HOL.

Acknowledgements. This work is funded by the EPSRC projects RoboCalc[4] (Grant EP/M025756/1) and CyPhyAssure (Grant EP/S001190/1), and the Royal Academy of Engineering.

References

1. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A., Timmis, J.: Automatic property checking of robotic applications. In: Intl. Conf. on Intelligent Robots and Systems (IROS), IEEE (2017) 3869–3876
2. Hoare, T., He, J.: Unifying Theories of Programming. Prentice-Hall (1998)
3. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing **21** (2009) 3–32
4. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)

[4] RoboCalc Project: https://www.cs.york.ac.uk/circus/RoboCalc/

5. Foster, S., Zeyda, F., Woodcock, J.: Unifying heterogeneous state-spaces with lenses. In: ICTAC. LNCS 9965, Springer (2016)
6. Hilder, J., Owens, N., Neal, M., Hickey, P., Cairns, S., Kilgour, D., Timmis, J., Tyrrell, A.: Chemical detection using the receptor density algorithm. IEEE Transactions on Systems, Man, and Cybernetics **42**(6) (2012)
7. Sherif, A., Cavalcanti, A., He, J., Sampaio, A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing **22**(2) (2010) 153–191
8. Santos, T., Cavalcanti, A., Sampaio, A.: Object-Orientation in the UTP. In: UTP 2006. Volume 4010 of LNCS., Springer (2006) 20–38
9. Foster, S., Thiele, B., Cavalcanti, A., Woodcock, J.: Towards a UTP semantics for Modelica. In: UTP. LNCS 10134, Springer (2016)
10. Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Information Processing Letters **135** (2018) 47–52
11. Bresciani, R., Butterfield, A.: A UTP semantics of pGCL as a homogeneous relation. In: IFM. LNCS 7321, Springer (2012)
12. Feliachi, A., Gaudel, M.C., Wolff, B.: Unifying theories in Isabelle/HOL. In: UTP 2010. Volume 6445 of LNCS., Springer (2010) 188–206
13. Blanchette, J.C., Bulwahn, L., Nipkow, T.: Automatic proof and disproof in Isabelle/HOL. In: FroCoS. Volume 6989 of LNCS., Springer (2011) 12–27
14. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Calculational verification of reactive programs with reactive relations and kleene algebra. In: Submitted to RAMICS 2018. Preprint at https://arxiv.org/abs/1806.02101.
15. Foster, S., et al.: Stateful-failure reactive designs in Isabelle/UTP. Technical report, University of York (2018) http://eprints.whiterose.ac.uk/129768/.
16. Zhan, N., Kang, E.Y., Liu, Z.: Component publications and compositions. In: UTP. Volume 5713 of LNCS., Springer (2008) 238–257
17. Cavalcanti, A., Woodcock, J.: A tutorial introduction to CSP in unifying theories of programming. In: PSSE. Volume 3167 of LNCS. Springer (2006) 220–268
18. Back, R., Wright, J.: Refinement calculus: a systematic introduction. Springer (1998)
19. Foster, S., Cavalcanti, A., Canham, S., Woodcock, J., Zeyda, F.: Unifying theories of reactive design contracts. Submitted to Theoretical Computer Science (Dec 2017) Preprint: https://arxiv.org/abs/1712.10233.
20. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM **18**(8) (1975) 453–457
21. Morgan, C., Vickers, T.: On the Refinement Calculus. Springer (1992)
22. Huffman, B., Kunčar, O.: Lifting and transfer: A modular design for quotients in Isabelle/HOL. In: CPP. Volume 8307 of LNCS., Springer (2013) 131–146
23. Miyazawa, M., Cavalcanti, A., Ribeiro, P., Li, W., Woodcock, J., Timmis, J.: Robochart reference manual. Technical report, University of York (June 2018) https://cs.york.ac.uk/circus/RoboCalc/assets/robochart-reference.pdf.
24. McEwan, A.: Concurrent Program Development in Circus. PhD thesis, Oxford University (2006)
25. Schäfer, T., Knapp, A., Merz, S.: Model checking UML state machines and collaborations. ENCTS **55**(3) (2001) 357 – 369
26. Miyazawa, A., Cavalcanti, A.: Refinement-oriented models of stateflow charts. Science of Computer Programming **77**(10-11) (2012)
27. Foughali, M., et al.: Model checking real-time properties on the functional layer of autonomous robots. In: ICFEM. Volume 10009 of LNCS., Springer (2016)