Guard extraction for modeling and control of a collaborative assembly station

Downloaded from: https://research.chalmers.se, 2023-09-30 13:41 UTC

Citation for the original published paper (version of record):
http://dx.doi.org/10.1016/j.ifacol.2021.04.053

N.B. When citing this work, cite the original published paper.
Guard extraction for modeling and control of a collaborative assembly station

Martin Dahl * Kristofer Bengtsson * Martin Fabian * Petter Falkman *

* Department of Electrical Engineering, Chalmers University of Technology, 412 96 Göteborg, Sweden (martin.dahl | kristofer.bengtsson | fabian | petter.falkman) @chalmers.se.

Abstract: A transition system represented by guards and actions can be amended by new guards computed in order to satisfy some specification. If the transition system is the result of composing smaller state machines, guard extraction can be used to put the new guards onto the guards the original state machines. Planning and verification can then be performed directly on the system with additional guards. In this paper we discuss the benefits of applying guard extraction as part of the modeling work in a modular control architecture, where reusable resources are composed using specifications. We show with an example from the development of an industrial demonstrator that even if the specification language is limited to invariant propositions, in practice many common safety specifications can be expressed when combined with a notion of which transitions are allowed to be restricted.

Copyright © 2020 The Authors. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0)

Keywords: Discrete event systems in manufacturing

1. INTRODUCTION

Automation systems are becoming increasingly complex, with automated planning algorithms (Ghallab et al., 2016), online robot motion planning (Alterovitz et al., 2016; Perez et al., 2016), cyber-physical automation systems (Monostori et al., 2016) and collaborative robots (Bauer et al., 2008).

Developing these systems with traditional engineering methods also becomes highly complex (Dahl et al., 2016), especially when developing online planning algorithms, that not only needs to be reliable and fast, but also adaptive and flexible. To aid in this trade-off, this paper presents an automated design process using guard extraction (Miremadi et al., 2011), that support the engineering work by simplifying control modeling, improving reusability, as well as improving quality by ensuring correctness.

The control system developed by this process is using a resource based control architecture defined in Dahl et al. (2019). In this architecture, resources are modeled in isolation in order to be reusable and guard extraction is one of the tools used to ensure safe interaction between them. The development of an industrial demonstrator is used as an example to motivate why even the simplest kind of control logic synthesis – automatically ensuring invariant propositions – can be an important tool, especially for modeling purposes. In practice, specifying behavior using invariant propositions is a common use case which covers a lot of safety requirements.

In the applied control architecture, the decision of which action to take next is based on the current valuations of the variables of the involved resources (e.g. sensor readings). As such, there are neither state nor event labels, instead the control system is interested only in the combinations of variable valuations that make up a certain state, and under which conditions the system can transit between these states.

The paper is structured as follows: starting with some preliminaries in Section 2, we then give a brief introduction to the control architecture our method is applied to in Section 3. The guard extraction procedure is discussed in Section 4. Section 5 details how the guard extraction procedure was used for modeling and control for an industrial demonstrator using the control architecture. In Section 6 concluding remarks are made.

2. PRELIMINARIES

Definition 1. A transition system (TS) is a tuple \( \langle S, \rightarrow, I \rangle \), where \( S \) is a set of states, \( \rightarrow \subseteq S \times S \) is the transition relation and \( I \subseteq S \) is the nonempty set of initial states. A state \( s \in S \) is a unique valuation of each variable \( \text{var} \) in the system. E.g. \( s = \langle v_1, \ldots, v_n \rangle \).

Variables have finite discrete domains, i.e. Boolean or enumeration types.

The transition relation \( \rightarrow \) can be created from transitions that modify the system state:

Definition 3. A transition has a guard predicate function and a set of action functions, which can update variables making up a state. Formally we encode transitions by explicitly writing the next values of each variable to primed...
mirrors of them that we refer to as next-state variables, but to
save space we often write the guard as a partial function
over the system’s variables (the rest are “don’t cares”) and
any variables not included in the transitions’ set of action
functions keep their current value.

Example: a system with two Boolean variables \( x \) and \( y \) has
\( S = \{ \langle \neg x, \neg y \rangle, \langle x, \neg y \rangle, \langle \neg x, y \rangle, \langle x, y \rangle \} \). A transition
with the guard predicate \( \neg x \) and the action function \( x \)
is encoded formally as \( \neg x \land x’ \land (y \leftrightarrow y’). \) In this example
\( \rightarrow \) is thus \( \{ \langle \neg x, \neg y \rangle, \langle x, \neg y \rangle, \langle \neg x, y \rangle, \langle x, y \rangle \} \).

3. CONTROL ARCHITECTURE

In this control architecture, the resources making up the
automation system are controlled by a central planner continuously
deciding which transitions to take. Resources are
modeled independently of each other, and composed by
1) adding specifications that ensure various properties as
well as 2) adding new transitions that capture new state
that emerge from their composition. In this work the focus is
on using guard extraction, as described in Section 4,
to ensure invariant propositions. The remainder of this
section will briefly define how the control architecture used
in the demonstrator is structured, starting with models of
the individual resources.

3.1 Resources

Devices and software algorithms in the system are modeled
as resources, which group their local state and discrete
descriptions of the tasks they can perform. The resource
state is encoded into variables of three kinds: measured
state, command state, and estimated state. Measured and
command states correspond to inputs and outputs to/from
the control system, while estimated states are internal
memory, usually to keep track of something.

Definition 4. A resource \( r_i \) is defined as \( r_i = \langle V_i^M, V_i^C, V_i^E, O_i, I_i \rangle \) where \( V_i^M \) is a set of measured state variables,
\( V_i^C \) is a set of command state variables, \( V_i^E \) is a set of
estimated state variables, and \( O_i \) is a set of generalized
operations defining the resource’s abilities. \( I_i \) is a set of
allowed initial states. Let \( V_i = V_i^M \cup V_i^C \cup V_i^E \).

3.2 Generalized operations

A generalized operation is defined to be able to express
both low-level ability operations and later on planning
operations, in a way that allows formal techniques for
verification and planning to be applied. Operations essen-
tially group predicates (Boolean functions) and transitions
that can update the local state of a resource, into logical
units that define the tasks that a resource can perform.
Predicates can, but do not need to, have names to clarify
their meaning. We write these as \( name: function \) where
\( name \) is an arbitrary name and \( function \) is the Boolean
function that defines the predicate.

Definition 5. A generalized operation \( o_j \) is defined as
\( o_j = \langle P_j, G_j, T^d_j, T^a_j, T^e_j \rangle \). \( P_j \) is a set of named predicates
over the state of the resource variables \( V_j \). \( G_j \) is a set
of un-named guard predicates over the state of \( V_j \). The
sets \( T^d \) and \( T^a \) define control transitions that update \( V_j \),
where \( T^d \) defines transitions that require (external from
the ability) deliberation and \( T^a \) defines transitions that
occur automatically whenever possible. \( T^e_j \) is a set of effect
transitions describing the possible consequences to \( V_j^M \)
of being in certain states. A transition \( t_{j} \in T^d \cup T^a \cup T^e \) has a guard predicate which can contain elements from \( P_j \) and
\( G_j \) and a set of actions that update the current state if
and only if the corresponding guard predicate evaluates to
true.

\( T^d_j, T^a_j, \text{and } T^e_j \) have the same formal semantics, but are
separated due to their different uses: deliberation transitions \( T^d_j \) require external input in order to be taken
by the control system (e.g. input from an online planning
system). Automatic transitions \( T^a_j \) are taken whenever possible
by the control system. Effect transitions \( T^e_j \) define how the
measured state is updated, and as such they are not used during control like the control transitions \( T^d_j \)
and \( T^a_j \). They are important to keep track of however, as
they are needed for online planning and formal verification
algorithms, as well as for simulation based validation.

It is natural to define when to take certain actions in
terms of what state a resource is currently in. To ease both
modeling, planning algorithms, and later on online moni-
toring, the guard predicates of the generalized operations
are separated into one set of named \( (P_j) \) and one set of
un-named \( (G_j) \) predicates. The named predicates can be
used to define the current state of an operation in terms of
the set of local resource states defined by this predicate.

3.3 Ability operations

The behavior of the resources in the system is modeled by
ability operations (abilities for short). Instead of having
unique predicate names for the states of an ability (e.g.
“is closing”), it is useful to define a number of “standard”
predicate names for abilities to ease modeling, reuse,
and support for online monitoring. In this work common
meaning is introduced for the following predicates: enabled
(ability can start), starting (ability is starting, i.e. a
handshaking state), executing (ability is executing, i.e.
waiting to be finished), finished (ability has finished), and
resetting (transitioning from finished back to enabled). For
most abilities, the transition between enabled and starting
will have an action in \( T^d \), while the transition from finished
to enabled will have an action in \( T^a \).

3.4 Planning operations

Control of an automation system can be abstracted into
performing operations (Lennartson et al., 2010). While
ability operations define the low-level tasks that different
resources can perform, planning operations model how
to make the system do something useful. The planning
operation \( k \) is defined as the generalized operation \( O_k \)
and the estimated state variable \( \hat{O}_k \) with the domain \( \{i, e, f\} \).
For the operation \( k \), \( P_k = \{ \langle init, \hat{O}_k = i \rangle, \langle executing, \hat{O}_k = e \rangle, \langle finished, \hat{O}_k = f \rangle \} \). \( P_k \) is used to keep track of the
current state of the operation, which is later used in
order to collect the current goal states of the system.
The operation has one deliberation transition \( t_k^d \in T_k^d \), with
the guard predicate \( init \land g_j \) (where \( init \) refers to \( init \)
in \( P_k \) ) and the action function \( \hat{O}_k := e \). This represents
starting the operation, where \( g_j \in G_k \) is an unnamed predicate defining the precondition of \( O_k \). Finishing the operation is defined by the automatic transition \( t^f_k \in T^a_k \) defined by the guard predicate \( \text{executing} \land g_k \) with the action function \( \tilde{O}_k := f \), where \( g_k \in G_k \) is a predicate defining the postcondition of \( O_k \). Operations do not have effect transitions: \( T^f_k = \emptyset \).

4. GUARD EXTRACTION

Invariant properties can be ensured by making sure that states where the properties do not hold cannot be reached, however, when the environment is part of the model this does not make much sense, as the environment cannot be restricted arbitrarily. But by combining the restriction on events with a notion of which \textit{control actions} are allowed to be restricted, a very natural way to model that the system is waiting for something external to the controller (i.e. some action) to happen, or that some action is important and should always need to be allowed to happen emerges. To model this, we borrow the concept of \textit{uncontrollability} from Supervisory Control Theory (SCT) (Ramadge and Wonham, 1987). In SCT, the goal is to find a supervisor that disables \textit{controllable} events generated by a plant in order to satisfy some specification, while \textit{uncontrollable} events should always be allowed. Furthermore, the resulting supervisor should be maximally permissive (Ramadge and Wonham, 1987), i.e. it should restrict the system as little as possible.

Consider a situation where a resource is shared between two processes. Most likely it is the \textit{acquisition} of the resource that we want to restrict, i.e. under which circumstances should we be allowed to acquire the resource, rather than controlling when to \textit{release} the resource. As such, the act of acquisition is naturally modeled as a controllable event, while the release of it is uncontrollable – it is in theory controllable but we want to design the system to release the resource as soon as we are done with it. In context of the control architecture defined in Section 3, the acquisition would be a \textit{deliberation} transition, and the release would be an \textit{automatic} transition.

Another situation is when we are waiting for a signal that occurs due to the controller interacting with the environment, for example, waiting for input from some sensor. As it is the environment that produces the effect, it cannot be controlled (by the control system). We would like to capture that we are in a waiting state however, and that from this state execution will continue, so we model transitioning between the waiting and done state as an uncontrollable event. This situation corresponds to the \textit{effect} transitions defined in Section 3.

4.1 Symbolic state representation

With a symbolic representation (Burch et al., 1992), sets of states are represented as Boolean functions. If a system is represented by the Boolean variables \( x \) and \( y \), the set of states \( \neg x \land \neg y \), \( x \land \neg y \), \( \neg x \land y \) can be represented by the Boolean function \( \neg (x \land y) \). The functions representing sets can then be manipulated by Boolean operations. Using the representation of a transition defined in Definition 3, a set of next states can be computed from a set of states by taking the conjunction of the function representing the set with the function representing the transition(s).

If the initial states of a TS are represented by the Boolean function \( \phi(s) \) and the transitions of the TS are represented by the function \( \gamma(x, x') \) (e.g. \( t_1(x, x') \lor t_2(x, x') \) where \( x, x' \) represent the current-state and next-state variables respectively), exploring the state space symbolically can then be done by finding the fix-point of:

\[
\begin{align*}
\phi_0(s) &= \phi(s) \\
\phi_{i+1}(s) &= \phi_i(s) \lor \exists s'. (\phi_i(s') \land \gamma(s', s))
\end{align*}
\]

Quantifying the source states out of the conjunction (i.e. \( \exists s'. (\phi_i(s') \land \gamma(s', s)) \)) above, the transitions, is commonly defined as the \textit{relational product} (\textit{replprod} below). Naively, this fix-point computation can be implemented as in Listing 1, where \texttt{initial} is a Boolean function representing the initial states, \texttt{trans} is a Boolean function representing the transitions, \texttt{vs} and \texttt{vsn} are the system variables where \texttt{vsn} represent the next-state variables. The \texttt{replace} function replaces each next-state variable with its corresponding state variable.

```plaintext
fn reachable(initial, trans, vs, vsn) {
  let mut r = initial;
  loop {
    let old = r;
    let new = replprod(old, trans, vs, vsn);
    let new = replace(new, vs, vsn);
    if old == r {
      break;
    }
  }
  return r;
}
```

Listing 1: Implementation of symbolic breadth first reachability.

Let us give an example to illustrate both the symbolic representation and how it automatically gives us guard extraction, if the set \( S_g \) of reachable and safe (safe as in not violating some specification) states is already computed.

Consider a TS represented with the variables \( x, y \), a transition relation containing \( \neg x \land x' \land (y \Leftrightarrow y') \) and \( \neg y \land y' \land (x \Leftrightarrow x') \), and \( \neg x \land \neg y \) as the initial state. If \( \neg (x \land y) \) is a specification and both transitions are controllable, it is easy to see that all safe and reachable states \( S_g \) can also be represented by \( \neg (x \land y) \). Any new guards for some transition \( t \) constrained by \( S_g \) can be extracted by taking the relational product of \( t \) and \( S_g \), where \( t \) is applied \textit{backwards}, i.e. with the current- and next-states of \( t \) swapped. The result of this computation is the symbolic representation of all source states from where taking \( t \) ends up in the safe set \( S_g \). Thus, for the transition that changes \( x \), any new guard(s) can be extracted by taking the transition \( \neg x \land x' \land (y \Leftrightarrow y') \) backwards, i.e. \( x \land \neg x' \land (y \Leftrightarrow y') \) from \( S_g \) by taking the conjunction of the two formulas, \( x \land \neg x' \land (y \Leftrightarrow y') \land \neg (x \land y) \) which gives \( x \land \neg x' \land (y \Leftrightarrow y') \land \neg y \). Quantify out \( x \) and \( y \) to get \( \neg x' \land \neg y' \), which only contains the next-values \( x' \) and \( y' \), which in this case represents the \textit{source states} of the original transition (as it was taken backwards). Substitute \( x' \) with \( x \), and \( y' \) with \( y \), and we get the final guard \( \neg x \land \neg y \),

...
of which $\neg x$ was the original guard of the transition and $\neg y$ is the added guard that forbids the system from violating the specification.

During implementation, ordered binary decision diagrams (Bryant, 1992) are commonly used to efficiently manipulate the Boolean expressions. In practice this results in the function representing $S_g$ containing terms that make no difference to the end result. For human understanding it is important to post-process the generated guards. See for example Miremadi et al. (2011).

What is left then is to find the set $S_g$ containing the safe states. Ideally, $S_g$ should be maximally permissive, i.e. contain as many states as possible, allowing for maximum freedom for the planning system.

### 4.2 Finding the safe states $S_g$

As stated, we want to find the maximally permissive set $S_g$ containing the states that are safe to be in (e.g. those that do not violate a specification). The problem we want to solve is called synthesis w.r.t. controllability in the SCT literature. We can solve this by computing the set of reachable states ($S_r$) and the set of forbidden states ($S_f$). Then $S_g = S_r \cap (S - S_f)$ contain all safe states. Listing 2, shows how to apply the symbolic reachability function to compute $S_g$, where $trans$ is a Boolean function representing all transitions of the system and $buc$ is a Boolean function representing the uncontrollable transitions backwards.

```plaintext
fn safe_states(initial, forbidden, trans, buc, vs, vsn) {
    let r = reachable(initial, trans, vs, vsn);
    let f = reachable(forbidden, buc, vs, vsn);
    return logical_and(r, logical_not(f));
}
```

Listing 2: Finding the safe states: `safe_states` returns the set $S_g$ symbolically.

### 4.3 Algorithm for resource composition

Visiting all reachable states of course puts a hard limit on the size of the systems that can be handled, even if they are nowadays rather large (e.g. more than $10^{20}$ states (Burch et al., 1992)). However, because the resources as defined in Definition 4 are independent of each other, finding $S_g$ can be done in a modular way.

For each specification, start by identifying which resources it involves by looking at the state variables that are referenced in the specification. Always group resources that share a variable. Construct a TS containing the state variables and transitions contained in each involved resource. Additionally, reverse (i.e. make them go backwards) all the automatic and effect transitions and add them to the set of backwards uncontrollable transitions ($buc$ in Listing 2).

The generated guard expressions are valid only within the states defined by $S - S_f$. As such, the allowed initial states of each resource need to be updated to also reflect the new forbidden states found for each specification. Updating of the individual resources with new guards and initial states is similar to the normalization procedure performed in Mohajerani et al. (2016).

### 5. APPLICATION: ASSEMBLY STATION

Fig. 1 shows an assembly station where a human operator and a collaborative robot work together to mount parts on a diesel engine. The collaborative robot hangs from the ceiling, together with some tools that can be attached via a tool changer attached to the robot. This section highlights how guard extraction was used as a modeling and control tool during the implementation of this demonstrator by showing how it was applied to model this changing of tools.

The robot (from here, the `ur10`), the tool changer (rsp), and a gripping tool (tool) are all modeled as individual resources.

To ease notation, we use a notation in which measured state variables are denoted with a subscript "$!\$", command state variables are denoted with a subscript "$\hat{\circ}$", and estimated state variables are denoted with a hat — see Table 1.

<table>
<thead>
<tr>
<th>v</th>
<th>measured state variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>command state variables</td>
</tr>
<tr>
<td>$\hat{v}$</td>
<td>estimated state variables</td>
</tr>
</tbody>
</table>

Table 1: Notation for the three different types of a resource state variable $v$.

The tool can open and close based on a digital output, $o_t$, which opens the gripper when high and closes it when low. The tool has sensors for knowing whether its opened ($o_t$) or closed ($c_t$). A resource $t$ modeling the tool is defined as $r_t = \{(c_t, o_t), \{c_t, o_t\}, \emptyset, O_d, I_d\}$. The allowed initial states $I_d$ are all states where the opened and closed sensors are not active at the same time. $O_d$ contains two abilities, close and open. These are expressed in terms of the tool resource state. Table 2 shows the transitions of the close and open abilities respectively. The type column denotes whether the predicate and action function make up a deliberation (d), automatic (a), or effect (e) transition. To save space, we allow to put named predicates in the...
tables. These are denoted by rows with ‘_’ for the action functions and the type.

<table>
<thead>
<tr>
<th>predicate</th>
<th>action functions</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled : ¬c1</td>
<td>o := true</td>
<td>d</td>
</tr>
<tr>
<td>executing : o1 ∧ ¬o2</td>
<td>o2 := true, c2 := false</td>
<td>e</td>
</tr>
<tr>
<td>finished : o1 ∧ o2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The transitions and named predicates making up the open (top), and close (bottom) abilities.

Next, the rsp resource is modeled. The rsp has two command variables, l ∈ {false, true} and u ∈ {false, true}, requesting whether it should be locked or unlocked. The rsp locking mechanism does not have a sensor, but it keeps its state even if powered off so as not to drop any currently held tools, and therefore the state of it must be estimated. To keep track of whether it is locked or not, the resource includes an estimated state variable \( \hat{l} \), as well as \( \hat{c} \), meaning that it is already locked, to put it in a known state, which also applies to unlocking.

The resource \( \hat{r} \) defining the rsp can then be defined as \( r_{\hat{r}} = \{0, \{u, \hat{u}\}, \{\hat{p}\}, \{\hat{t}\}, I_r \} \), where \( I_r \) contains all states where \( \hat{l} = \text{unknown} \). Table 3 shows the abilities for locking and unlocking the tool changer.

<table>
<thead>
<tr>
<th>predicate</th>
<th>actions functions</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled : ( l \neq true )</td>
<td>( l := true, u := false, \hat{l} := true )</td>
<td>d</td>
</tr>
<tr>
<td>finished : ( l = true )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: The transitions and named predicates making up the lock and unlock abilities of the rsp resource.

The \( ur10 \) can be in one of \( P \in \{p0, p1\} \) predefined poses. A resource \( u \) is defined for the \( ur10 \) as follows: \( r_u = \langle \{p_1, m_1\}, \{gp\}, \{\hat{p}\}, O_u, I_u \rangle \), where \( p_1 \) is the measured robot position, \( m_1 \in \{false, true\} \) is a measure of whether the robot is moving or stationary, \( gp \) is its goal position and \( \hat{p} \) is the last known observed position. Table 4 shows the move_to_p0 ability of the \( ur10 \) resource. Corresponding abilities exist for each target position of the robot.

<table>
<thead>
<tr>
<th>predicate</th>
<th>actions functions</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>enabled : ( p_1 \neq p_0 \land gp \neq p_0 )</td>
<td>( gp := p_0, m_1 := true )</td>
<td>d</td>
</tr>
<tr>
<td>starting : ( p_1 \neq p_0 \land gp = p_0 \land m_1 )</td>
<td>( m_1 := true )</td>
<td>e</td>
</tr>
<tr>
<td>executing : ( gp = p_0 \land p_1 \neq p_0 \land m_1 )</td>
<td>( p_1 := p_0, m_1 := false )</td>
<td>e</td>
</tr>
<tr>
<td>finished : ( gp = p_0 \land \hat{p} := p_0 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: The transitions and named predicates making up the move_to_p0 ability of the \( ur10 \) resource.

In order to model the interaction between the tool and the \( ur10 \), an estimated state variable is introduced, \( \hat{t} \), is updated by the transitions defined in Table 5. When the robot is at \( p_0 \) and the rsp is unlocked, the tool is considered to be at the home position and when the robot is at \( p_0 \) and the rsp is locked, the tool is considered to be attached to the robot.

<table>
<thead>
<tr>
<th>predicate</th>
<th>actions functions</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{t} = robot \land p_1 = p_0 \land \neg \hat{l} )</td>
<td>( \hat{t} := \text{home} )</td>
<td>a</td>
</tr>
<tr>
<td>( \hat{t} = \text{home} \land p_1 = p_0 \land \hat{l} )</td>
<td>( \hat{t} := \text{robot} )</td>
<td>a</td>
</tr>
</tbody>
</table>

Table 5: Two automatic transitions for keeping track of the tool position.

It is now possible to define two planning operations, one for taking the tool and one for putting it back, while also moving the robot to \( p_1 \). We define TakeTool to have the preconditions \( \hat{t} = \text{home} \) and the goal state \( \hat{t} = \text{robot} \land p_1 = p_1 \), as well as LeaveTool with the precondition \( \hat{t} = \text{robot} \) and the goal state \( \hat{t} = \text{home} \land p_1 = p_1 \). The desired outcome is described in Fig. 2, which shows two possible plans for executing the planning operations.

Fig. 2. Two plans arising from the intent of taking and putting back the tool illustrated beside the respective planning operation. In the TakeTool case, the \( rst \) is initially unlocked, the tool is closed (hanging at the stand) and the robot is at \( p_1 \). In the LeaveTool case, the \( tool \) is initially closed and attached to the robot (i.e. \( rst \) is locked), the \( ur10 \) is in \( p_1 \), and the \( rst \) is locked.

While it is easy to model in the level of abstraction provided by the planning operations, to get the behavior shown in Fig. 2, additional specifications are needed in order to ensure proper interaction between the resources.

5.1 Ensuring invariant propositions via additional guards

If the tool is attached to the robot, it should not be possible to move the robot between poses \( p_0 \) and \( p_1 \) unless the tool is open. This can be expressed as specification A: \( \hat{t} = \text{robot} \land \hat{p} = p_1 \land ur10\text{-move_to_p1}\text{-executing} \Rightarrow tool\text{-open}.finished \) as well as specification B: \( \hat{t} = \text{robot} \land \hat{p} = p_1 \land ur10\text{-move_to_p0}\text{-executing} \Rightarrow tool\text{-open}.finished \).

Similarly, if the tool is at the tool stand, it should not be possible to move from pose \( p_1 \) to \( p_0 \) unless the \( rst \) is unlocked, or the tool changer will collide with the tool. This can be expressed as specification C: \( \hat{t} = \text{home} \land ur10\text{-move_to_p0}\text{-executing} \Rightarrow \text{rstunlock}.finished \). If the robot is holding the tool, the \( rst \) is not allowed to unlock to release the tool unless the robot is at \( p_0 \) and the gripper has closed (thus holding on to the tool stand). This is specification D: \( \hat{t} = \text{robot} \land \text{rstunlock}.finished \Rightarrow p_1 = p_0 \land tool\text{-close}.finished \).

Finally, if the tool is at the tool stand, the gripper must not be allowed to open. E: \( \hat{t} = \text{home} \Rightarrow \text{tool\text{-close}.finished} \).
5.2 Guard extraction results

To perform guard extraction by reverse reachability from forbidden states, the negation of each specification are disjuncted to form a Boolean function representing all forbidden state combinations. Then the guard extraction algorithm described previously can be applied to the TS created from the resources, their initial states and the set of forbidden states. The generated guards can be seen in Table 6.

<table>
<thead>
<tr>
<th>Ability</th>
<th>New guard</th>
</tr>
</thead>
<tbody>
<tr>
<td>ur10.move_to_p0</td>
<td>¬p0 ∧ ¬p0 ∧ ¬t ∧ l ∧ l ∧ l = robot</td>
</tr>
<tr>
<td>ur10.move_to_p1</td>
<td>p0 ∧ p0 ∧ t ∧ l ∧ l ∧ l = robot</td>
</tr>
<tr>
<td>rep.lock</td>
<td>l ∧ l ∧ l = home ∨ ¬o ∧ o ∧ l ∧ l ∧ l = robot</td>
</tr>
<tr>
<td>rep.unlock</td>
<td>¬o ∧ l ∧ l = home ∨ ¬o ∧ o ∧ l ∧ l ∧ l = robot</td>
</tr>
<tr>
<td>tool.open</td>
<td>l ∧ l = robot</td>
</tr>
<tr>
<td>tool.close</td>
<td>p0 ∧ ¬p0 ∧ p0 ∧ ¬p0 ∧ p0 ∧ ¬p0 ∧ p0 ∧ ¬p0 ∧ p0 ∧ ¬p0 = robot</td>
</tr>
</tbody>
</table>

Table 6: Produced guards for the deliberation transitions of respective ability.

With the new guards (and the new initial states of the resources, not shown here), the set of safe states Ss contains 1056 states. The computed guards, while certainly not impossible to understand, are arguably more complex than the specifications A-E provided above, which each solve a distinct problematic interaction. With the computed guards added back to the resource’s deliberation transitions, the control system will now produce plans according to Fig. 2. While the above example may seem trivial, it is actually a quite tricky situation due to the interconnected resources.

6. CONCLUSION

At the surface, some specifications seem very easy to get rid of by simply modifying the involved resources. However, one needs to remember that the discrete descriptions of the resources are just one part of a larger collection of driver code, simulation representations etc. Ideally the resource definitions should not need to be changed.

Additionally, it may seem limiting only to be able to specify invariant propositions, however when combined with the notion of uncontrollable transitions, we have found that in practice, this comes a long way towards expressing the most common safety specifications.

The method outlined here does not take into account global specifications – each specification is handled only for the involved resources (in the example above the three resources are interconnected and treated as one). This means that other types of verification need to be performed after the guard extraction has been performed. For example, it is essential to check that whenever a precondition of a planning operation is satisfied, there must exist at least one path leading to the goal state of the operation. This highlights the fact that the guard extraction is just one of many tools that can be used to ensure high quality automation solutions. Perhaps the biggest advantage to applying guard extraction lies in its simplicity – because it only modifies the guards of the original resources, it can easily coexist with both traditional model checking and planning systems.

In this work the resources and specifications are grouped in a very conservative way. Future work will involve looking into better ways to create the sub-problems solved to find the set of safe states by inspecting the transitions involved.

REFERENCES


