OPTIMIZING SLICING OF FORMAL SPECIFICATIONS BY DEDUCTIVE VERIFICATION

INGO BRÜCKNER
Universität Oldenburg, Department für Informatik
26111 Oldenburg, Germany
ingo.brueckner@informatik.uni-oldenburg.de

BJÖRN METZLER
Universität Paderborn, Institut für Informatik
33098 Paderborn, Germany
bmetzler@uni-paderborn.de

HEIKE WEHRHEIM
Universität Paderborn, Institut für Informatik
33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Slicing is a technique for extracting parts of programs or specifications with respect to certain criteria of interest. The extraction is carried out in such a way that properties as described by the slicing criterion are preserved, i.e., they hold in the complete program if and only if they hold in the sliced program. In verification, slicing is often employed to reduce the state space of specifications to a size tractable by a model checker.

The computation of specification slices relies on the construction of dependence graphs, reflecting (at least) control and data dependencies in specifications. The more dependencies the graph has, the less removal of parts is possible. In this paper we present a technique for optimizing the construction of the dependence graph by using deductive verification techniques. More precisely, we propose a technique for showing that certain control dependencies in the graph can be eliminated. The technique employs small deductive proofs of the enabledness of certain transitions. Thereby we obtain dependence graphs with less control dependencies and as a consequence smaller specification slices which are an easier target for model checking.

Key words: Program Slicing, Integrated Formal Specifications, Exact Reduction, Deductive Verification, Model Checking.

ACM CCS Categories and Subject Descriptors: Software Engineering (D2), Requirements/Specification (D2.1), Software/Program Verification (D2.4).

1. Introduction

Slicing (see [18]) originates from the area of program analysis [13] where it has first been employed for debugging programs. The initial idea of
Weiser [21] was to extract that part of a program which may influence the value of a variable at a certain program point (the slicing criterion). Since then, the application of slicing techniques has spread to a variety of fields, in particular also to verification [8]. The purpose of applying slicing in a verification process is to construct a (hopefully small) part of the specification/program on which a certain temporal logic property holds if and only if it holds for the complete specification. Slicing can thus be seen as one out of a number of techniques for fighting the state explosion problem (together with abstraction [5], partial-order reduction [14], heuristic search [6] etc.).

Slicing usually starts with the construction of a dependence graph [11] which reflects the dependencies among entities in the specification (or statements in the program). These dependencies are used to compute those parts of a specification which might affect a certain property. The less dependencies we have, the smaller the slice can get. However, since the slice should precisely reflect the property under interest, dependencies can only be removed if entities are really independent. It has been shown that the computation of optimal slices is undecidable in general and can become PSPACE- or NP-complete for certain classes of programs [12].

In this paper, we are concerned with slicing of specifications written in an integrated notation (called CSP-OZ [7]), combining two well known formal specification techniques: CSP [9] for behavioural aspects of systems and Object-Z (OZ) [16] for data aspects of systems. This combination allows us to conveniently specify systems by different views: one view specifying orderings of operations, parallelism and communication among components (by means of CSP), and another view focusing on data and their operations (Object-Z). The reason for applying slicing to such specifications is that their state spaces usually grow very fast, in particular due to the data coming from the Object-Z part. Still, we are interested in model checking CSP-OZ specifications (for instance using a technique proposed in [10]) and thus need reduction techniques. Slicing methods for Object-Z alone and for CSP-OZ wrt. slicing criteria formulated in temporal logic have been proposed in [2] and [1]. They include the construction of a specification dependence graph (SDG), which comprises — similar to a program dependence graph — all relevant kinds of dependencies between specification elements such as control or data dependencies.

In these previous approaches the specification dependence graph is solely computed from the syntactic description of the specification. However, as case studies conducted so far have shown, the SDG contains a certain amount of control dependencies which are — when looking at the semantics of specifications — unnecessary, i.e., the syntactically derived control dependencies overapproximate the actual semantic dependencies. The present paper thus aims at a further improvement of slicing by combining the construction of the SDG with small deductive proofs which can be used to find out irrele-
vant control dependencies. The proofs have to show enabledness of Object-Z operations under assumptions on the CSP part and are inspired by a technique for showing deadlock freedom of integrated specifications developed in [19]. Based on these additional arguments, some control dependencies can be eliminated from the SDG. This can in consequence lead to smaller and more precise specification slices. Our main contribution here is to show, to which specification parts such deductive verification techniques can be applied and how the necessary arguments can be found and used to optimize slicing.

The paper is structured as follows. Next, we introduce CSP-OZ by an example specification which will serve as an illustration of the main results in the next sections. We also describe the operational semantics of our formalism. Section 3 introduces program slicing for CSP-OZ specifications. We will compute the specification dependence graph (SDG) of our example specification and explain how the computation of a program slice wrt. a certain verification property is accomplished. In Section 4, we describe our way of deductively showing enabledness of operations and we apply the results to our example. Subsequently we combine both techniques, illustrate the improvement by means of our example and show correctness of this optimization process. The last section concludes.

2. Specifying with CSP-OZ

We introduce CSP-OZ by specifying an example elevator system as depicted in Figure 1. The specification’s CSP part defines the elevator’s dynamic behaviour. This is done by introducing a set of channels that define the elevator’s interface for communication with its environment. How this communication looks like is defined in two CSP processes in terms of the order of events the elevator might be engaged in: first (CSP process equation \( \text{main} = \ldots \)), the passenger requests to be delivered to a certain floor (event \( \text{req} \), connected by the CSP prefix operator \( \rightarrow \) to the remainder of the \( \text{main} \) process), upon which the door \( \text{closes} \) (another application of the prefix operator) and the elevator starts to Work (a CSP process call, leading to the second process equation). Subsequently, in the definition of process Work, the CSP external choice operator models the fact that at this point the elevator has two possible behaviours: either it moves, followed by another call of process Work, or it stops, followed by opening its doors and returning to the initial \( \text{main} \) process, i.e., a restart of the elevator system. Which one of these two alternative behaviours are pursued by the elevator system, depends on the Object-Z part of the specification that follows beneath the CSP process definitions.

The specification’s Object-Z part defines the system’s state space, its initial configuration and operations on it by so called schemas. Each schema
Status ::= open | closed

Elevator
chan req : [n? : N]
chan close, move, open, stop
main = req → close → Work
Work = (move → Work) □ (stop → open → main)

| rFloor : N | rFloor = 0 |
| aFloor : N | aFloor = 0 |
| door : Status | door = open |

com_req
Δ(rFloor); n? : N
aFloor = rFloor
rFloor' = n?

com_close
Δ(door)
| door = open |
| door' = closed |

com_move
Δ(aFloor)
(aFloor < rFloor ∧
aFloor' = aFloor + 1) ∨
(aFloor > rFloor ∧
aFloor' = aFloor - 1)

com_open
Δ(door)
| door = closed |
| door' = open |

com_stop
| aFloor = rFloor |

Fig. 1: Elevator specification

consists of two parts: the upper part may contain a Δ-list of variables that are modified in the lower part. In schema \texttt{com\_move} for instance, \(\Delta(aFloor)\) indicates that variable \texttt{aFloor} is modified when event \texttt{move} takes place. The variable’s value before execution of the event is referred to by the unprimed variable \(aFloor\), while its value afterwards is referred to by its primed version \(aFloor'\). Additionally, the upper part of a schema may contain a list of input and output parameters (decorated with \(?\) and \(!\), respectively), that are used in the lower part for incoming or outgoing communications (e.g. \(n? : N\) in schema \texttt{com\_req} indicates that the elevator’s environment will provide an input \(n?\) of type \(N\) when event \texttt{req} takes place). The lower part can contain a set of predicates over primed and unprimed variables and over input and output parameters. These predicates determine on the one hand
whether the associated event is enabled or not and define on the other hand the \textit{effect} of the associated event on the state space.

Whether an event is enabled or not can be determined by computing its precondition, denoted by the precondition operator \( \text{pre} \):

$$\text{pre} \ Op = \exists State' \bullet Op \setminus \text{outputs}$$

Applied to a schema \( Op \) with a set of output parameters \( \text{outputs} \), this operator yields the predicates that form the schema’s preconditions. When one of these predicates is unsatisfied, its associated event is blocked, otherwise it is enabled. In schema \texttt{com\_open}, for instance, predicate \( \text{door} = \text{closed} \) defines a precondition that blocks event \( \text{open} \), unless the \( \text{door} \) is \( \text{closed} \).

The \textit{effect} of an event on the state space is defined by relating the current variable values (referred to by unprimed variables) with variable values after the event’s execution (referred to by primed variables). In schema \texttt{com\_open}, for instance, predicate \( \text{door}' = \text{open} \) defines \( \text{door} \) to be \( \text{open} \) after event \( \text{open} \) has taken place.

\section*{2.1 Specification Semantics: Labelled Kripke Structures}

The semantics of CSP-OZ classes can be described via labelled Kripke structures. In contrast to ordinary Kripke structures, transitions are labelled with events. This allows us to use temporal logics for property specification which not only talk about states of the system but also about execution of events.

\begin{definition} (Labelled Kripke structures) 
Let \( AP \) be a non-empty set of atomic propositions, \( E \) an alphabet of events (consisting of method names plus values of parameters).

An (event-)labelled Kripke structure \( K = (S, S_0, \rightarrow, L) \) over \( AP \) and \( E \) consists of a finite set of states \( S \), a set of initial states \( S_0 \subseteq S \), a transition relation \( \rightarrow \subseteq S \times E \times S \) and a labelling function \( L : S \rightarrow 2^{AP} \).
\end{definition}

For our elevator example, atomic propositions might for instance be \( \text{door} = \text{open} \) or \( \text{aFloor} = 0 \). As we will see next, the labelled Kripke structure for an entire CSP-OZ class is derived in two steps: in the first step we separately compute the semantics for the Object-Z part and the CSP part. Afterwards we combine the resulting Kripke structures into one via parallel composition and synchronization on events.

Before we define the semantics, we start with some necessary definitions. In the following, let \( Z = (\text{State}, \text{Init}, (\text{com}_m)_{m \in \text{Methods}}) \) be the Object-Z part of a CSP-OZ class \( C \) with \( \text{State} \) referring to the class’ state schema, \( \text{Init} \) to its initial state schema and \( \text{Methods} \) to the set of all methods used in the Object-Z part. Let \( \text{main} \) be the CSP part of a CSP-OZ class \( C \). The alphabet that we use here is the set \( \text{Events} = \{m.\text{i}.o \mid m \in \text{Methods}, i \in \text{in}(m), o \in \text{out}(m)\} \) of all CSP events, consisting of a channel (which is a
method of the Object-Z part) and values for input and output parameters with \( \text{in}(m) \) and \( \text{out}(m) \) respectively being the set of all input and output parameters of \( m \). We assume the CSP part to be \textit{data independent} in that it neither restricts nor uses values of parameters.

\textbf{Definition 2. (Kripke structure semantics of the Object-Z part)}

The Kripke structure semantics of the Object-Z part is defined as the labelled Kripke structure

\[
K^{\text{OZ}} = (\text{State}, \text{Init}, \rightarrow^{\text{OZ}}, L^{\text{OZ}})
\]

with the transition relation \( \rightarrow^{\text{OZ}} = \{(z, m, i, o, z') | \text{com}_m(z, i, o, z')\} \) and the labelling function \( L^{\text{OZ}} \) mapping each state onto the set of atomic propositions over the Object-Z state space that are valid in this state.

Note that we use \textit{blocking semantics} here: the precondition of an operation (\( \text{pre} \)) acts as a guard for its execution. Instead of \( z \xrightarrow{e}^{\text{OZ}} z' \) we also write \( z \xrightarrow{e}^{\text{OZ}} z' \) and instead of \( z \xrightarrow{e_1 \ldots e_n}^{\text{OZ}} z' \) we also write \( z \xrightarrow{(e_1, \ldots, e_n)}^{\text{OZ}} z' \). For any \( z \in \text{State} \) we furthermore define:

\[
\begin{align*}
& \circ z \xrightarrow{e}^{\text{OZ}} :\Rightarrow \exists z' \in \text{State} \bullet z \xrightarrow{e}^{\text{OZ}} z', \\
& \circ z \xrightarrow{\tau}^{\text{OZ}} :\Rightarrow \exists z' \in \text{State} \bullet z \xrightarrow{\tau}^{\text{OZ}} z', \\
& \circ \text{init}(z) := \{ e \in \text{Events} | z \xrightarrow{e}^{\text{OZ}} \}.
\end{align*}
\]

The semantics of the CSP part is defined as the labelled Kripke structure \( K^{\text{CSP}} = (\text{CSP}, \{ \text{main} \}, \rightarrow^{\text{CSP}}, L^{\text{CSP}}) \) with CSP denoting the set of all CSP terms, \( \text{main} \) being the only initial CSP term, \( \rightarrow^{\text{CSP}} \) being defined according to the operational semantics of CSP [15] and with the labelling function \( L^{\text{CSP}} \) mapping each CSP process onto the set of all atomic propositions over the Object-Z state space since the CSP part makes no restrictions on values of attributes of the class.

\textbf{Definition 3. (Kripke structure semantics of a CSP-OZ class)}

The Kripke structure semantics of a CSP-OZ class \( C \) is the parallel composition of the semantics of the Object-Z part and the CSP part: \( K = (\text{State}_C, \text{Init}_C, \rightarrow, L_C) \) with \( \text{State}_C = \text{State} \times \text{CSP} \), \( \text{Init}_C = \text{Init} \times \{ \text{main} \} \), \( L_C(z, P) = L^{\text{OZ}}(z) \cap L^{\text{CSP}}(P) \) and

\[
\quad \rightarrow = \{(z, P, e, (z', P')) | (e \neq \tau, P \xrightarrow{e}^{\text{CSP}} P', z \xrightarrow{e}^{\text{OZ}} z') \lor (e = \tau, P \xrightarrow{\tau}^{\text{CSP}} P', z = z')\}
\]

The relation \( \Rightarrow \) for traces is defined accordingly. The special symbol \( \tau \) describes an internal event of the CSP process. It is not observable in any trace and not part of the set \( \text{Events} \).
For describing properties of CSP-OZ classes we can now use any stuttering invariant temporal logic which can be interpreted on labelled Kripke structures. The main purpose of slicing in the context of verification is to determine which part of a specification actually has to be considered when checking for a given property, i.e., whether it is possible to check the property on a reduced specification such that the following holds (where $C \models \varphi$ expresses that the formula $\varphi$ holds on the Kripke structure of the specification $C$ and $C_{\text{red}}$ denotes the reduced specification):

$$C \models \varphi \iff C_{\text{red}} \models \varphi$$

In the next section we will show how to compute such a reduced specification that will hopefully be an easier target for verification than the full specification.

3. Slicing

To compute the slice of a specification in the context of verification means to compute a reduced specification that exhibits — from the point of view of the verification property — the same behaviour as the original specification. Our approach to achieve this goal for CSP-OZ specifications [1] consists of two main steps which are explained in the following subsections.

3.1 Specification Dependence Graph

First, the specification is analyzed with respect to the control and data dependencies it contains, resulting in a specification dependence graph (SDG). In preparation for the construction of the SDG we first construct the specification’s control flow graph (CFG) which represents the execution order of the specification’s schemas according to the specification’s CSP processes. Starting with the start.main node, its nodes and edges are derived from the syntactical elements of the specification’s CSP part, based on an inductive definition for each CSP operator. Nodes either correspond to schemas of the Object-Z part (like node req for schema com.req) or to operators in the CSP part (like node extchoice for operator $\Box$).\(^2\) We refrain from giving a precise definition here.

Based on such a CFG, we then proceed to the construction of the SDG which has the same set of nodes as the CFG, connected by newly introduced edges: control dependence edges represent the fact that a source node

\(^1\) The requirement of stuttering invariance is due to the fact that our goal is to reduce the specification and accordingly the paths over which the logic will be interpreted. Therefore the logic should not be able to precisely speak about particular steps of the system.

\(^2\) Note, that we assume each syntactical CSP element and each associated CFG node to have a unique name. This can, for example, be achieved by extending their names by an index that represents the position of their textual occurrence inside the specification. For sake of clarity we omit these indices here.
determines whether control flow reaches a target node or not, while data dependence edges represent the fact that a variable modification in a source node might reach a target node.

Regarding control dependence edges we distinguish a number of different types such as for instance the following one which is of particular importance for our example specification:

- Control dependence due to nontrivial precondition exists between a node and its CFG successor iff the precondition of the node's schema is non-empty (i.e., not equivalent to true).

An example for this type of control dependence edge is the one between nodes `req` and `close`: since schema `com_req` has a non-trivial precondition \((aFloor = rFloor)\), `req` is source of a control dependence edge leading to `req`'s CFG successor `close`.

Regarding data dependence edges we also distinguish different types with only the following one occurring in our example specification:

- Direct data dependence exists between two nodes iff the source node modifies a variable that is referenced by the target node and there is a CFG path from the source node to the target node without any further modification of the relevant variable along the path.

An example for this type of data dependence edge is the one between nodes `req` and `stop`: since schema `com_req` modifies variable `rFloor` which is referenced in schema `com_stop` and there are no further modifications along the CFG path from `req` to `stop`, `req` is source of a data dependence edge leading to `stop`.

The overall result of these definitions for the `Elevator` specification is the specification dependence graph as depicted in Figure 2. Note that this first step of constructing the specification dependence graph is completely independent of the property to be verified.

### 3.2 Slice

Once the SDG is constructed, it can be used for slicing the specification with respect to various verification properties (e.g. temporal logic formulae). Based on the property to verify, an initial set of SDG nodes is determined that directly influence the verification property. This set of nodes represents the *slicing criterion* that serves as the starting point for a backwards reachability analysis of the SDG. Regarding our example specification, the verification property might for instance be \(\Box \Diamond (aFloor = rFloor)\), expressing that the designated floor is always eventually equal to the current floor. From this formula we then derive the initial set of nodes \{`req`, `move`\} which are nodes whose associated events are directly mentioned in the formula (none in our example) and nodes that directly modify variables occurring in the formula, i.e., either `aFloor` or `rFloor`. 
The next step is to identify all nodes that cannot be reached backwards via control or data dependence edges from the initial sets of nodes, since these are nodes without any direct or indirect influence on the verification property. Thus the specification elements that are represented by these nodes can — in the final step of the slicing approach — safely be removed from the original specification without changing its semantics with respect to the verification property. The advantage is that now the sliced specification can be analyzed instead of the full specification, and although control and data state space have been reduced, the model checking result holds for both specifications: it has been shown that the verification property is satisfied by the reduced specification if and only if it is satisfied by the full specification [1].

3.3 Problem: Control Dependence Edges

Applied to the example specification, our slicing algorithm unfortunately does not achieve any reduction, regardless of which verification property serves as the slicing criterion. The reason for this can easily be seen in the SDG: regardless of which node we take as the starting point, we can always reach all other nodes backwards via control dependence edges, even if the graph would contain no data dependence edges at all. The reason for this is our somehow coarse definition of control dependence edges that is only based on the syntactic examination of the specification: unless an event’s precondition is trivially true (i.e., it has no precondition), its associated node is always source of a control dependence edge. More precisely, even if an event’s precondition is — on the semantic level — obviously satisfied
at any possible point of execution of the event, its associated node will be source of a control dependence edge. Thus, what we need here is a way to identify these non-trivial (but always satisfied) preconditions, such that we can subsequently eliminate the control dependence edges of their associated nodes, while the resulting slice remains correct. In the following sections we will sketch how to achieve exactly this by applying deductive verification techniques.

4. Identifying Non-blocking Events

In the previous section we identified the SDG control dependence edges to be responsible for the failing of our slicing algorithm. These edges represent the fact that their source nodes determine whether control flow reaches the target node.

From nodes representing events a control dependence edge can only originate if the associated schema has a non-trivial precondition, i.e., a precondition not equivalent to true such that the event can block when the precondition is not satisfied. But even if the precondition of a source node is non-trivial, it may still always be satisfied when control flow reaches the source node. In this case, the source node does not control the execution of the target node, i.e., the control dependence edge can be eliminated. However, our slicing algorithm does only consider trivial preconditions.

The following question arises: based on a CSP-OZ specification, how can we ensure that a precondition of a specific event is always satisfied at its execution and how can we prove this?

The solution to this question can be described as follows: let us consider a source node for a control dependence edge representing an Object-Z event $e$. We show that after executing an arbitrary trace leading from the start of the CSP part to the uniquely identified position of this event, its precondition is satisfied.\(^3\) In this case, the node $e$ does not control any subsequent nodes of the SDG since it is always possible to execute the operation $e$.

To describe traces leading to the occurrence of an event, we assume the CSP part to have process identifiers $P_0, \ldots, P_n$ and describe it as a set of process equations. For simplicity, we write $P_0$ instead of main:

\[
\begin{align*}
P_0 &= \ldots \rightarrow P_{i_1} \\
P_1 &= \ldots \rightarrow P_{i_2} \\
\vdots \\
P_n &= \ldots \rightarrow P_{i_n}
\end{align*}
\]

\(^3\) To specify the position, we assume only one occurrence of each event in the CSP part. In case one event occurs more than one time, an indexing on this event can be realized to guarantee a unique identification.
A trace starting in $P_0$ leading to a specific event passes several process calls. It can be split into subtraces leading from one process call to the next (these will be called $maxTraces$ in Definition 4):

$$P_0 \xrightarrow{tr} \text{CSP } P' \text{ with } P' \xrightarrow{e} \text{CSP } P''$$

can be viewed as

$$P_0 \xrightarrow{tr_0} \text{CSP } P_{r_1} \xrightarrow{tr_1} \cdots \xrightarrow{tr_k} \text{CSP } P_{r_k} \xrightarrow{tr_k} \text{CSP } P'$$  \hspace{1cm} (1)$$

if $tr = tr_0 \circ \cdots \circ tr_k$.

To establish that after (an arbitrary) trace $tr$ the precondition of the event $e$ is satisfied, we assign predicates on the state space of the Object-Z part to each process equation. A predicate must hold at each call of its associated process. We need to establish continuity in the following sense: the execution of each subtrace leads to a state that satisfies the predicate of the next process. Assume that the predicate $p_i$ for a process $P_i$ holds and assume execution of a trace leading to the call of a process $P_j$. Then the predicate $p_j$ must be satisfied. Finally, when reaching the event $e$, its precondition $\text{pre } e$ must hold:

![Diagram](image)

Our approach uses deductive verification techniques and is similar to the idea that [19] applied to prove deadlock freedom of CSP $\parallel$ B specifications, where the existence of control loop invariants (CLIs$^4$) leads to the result of deadlock freedom of the complete specification. The CLIs are our predicates on the state space of the specification, which are associated with process calls.

In the following subsection we identify conditions on a set of CLIs. Predicates satisfying these conditions yield that the precondition of a certain event is always satisfied. In this case an elimination of the respective control dependence is possible.

$^4$ The term control indicates that the CSP part controls the order of execution of the Object-Z schemas.
4.1 Weakest Liberal Precondition

As already mentioned we need to ensure that the validity of our CLIs is maintained during progress of execution along the CSP processes. This will be established by using a predicate on the Object-Z state space describing the weakest liberal precondition of the respective CLI wrt. associated traces. A weakest liberal precondition semantics for Object-Z can for instance be found in [3]. We use it to model the weakest predicate holding prior to execution of a trace that leads to the next process call with its associated CLI being valid.

In the following, we restrict the CSP part to not include sequential composition.\(^5\) This restriction is feasible since any CSP process containing sequential composition can be transformed into an equivalent process without it.

We use the following notations: let \( PId \) be the set of all process identifiers used in the CSP part with \( PId \cap Methods = \emptyset \). The mapping \( \text{traces}(P) \) computes the set of all possible traces of a CSP process \( P \). Finally, the set

\[
\text{initials}(P) = \{ e \in Events \mid (e) \in \text{traces}(P) \}
\]

describes all possible events that the CSP process \( P \) is able to perform initially. The internal event \( \tau \) is not observable, so any \( e \in \text{initials}(P) \) may be executed after an arbitrary sequence of \( \tau \).

First, we need two definitions describing the split of a trace as illustrated in trace (1) on the previous page: we define traces leading from one to the next recursive call of a process. After the last recursive call, we describe the traces leading from this process to the respective event:

**Definition 4.**

Let \( P \) and \( Q \) be two CSP processes. The set of maximal traces leading from \( P \) to \( Q \) without any other call of a process identifier inbetween is defined as

\[
\text{maxTraces} \ (P, Q) = \{ tr \in \text{traces} \ P[\text{STOP}/PId] \mid P \xrightarrow{\text{tr}}_{\text{CSP}} Q \}
\]

The set of traces before a fixed occurrence of a certain event \( e \) inside of \( P \) is defined as

\[
\text{preTraces} \ (P, e) = \{ tr \in \text{traces} \ P[\text{STOP}/PId] \mid \exists Q \cdot P \xrightarrow{\text{tr}}_{\text{CSP}} Q \land e \in \text{initials}(Q) \}
\]

\( P[\text{STOP}/PId] \) describes the process \( P \) with every occurrence of a process identifier replaced by \( \text{STOP} \). By using process replacement, we are able to

---

\(^5\) The elimination of sequential composition leads to the fact that the call of a process is always the last action inside (possibly a branch of) a process. This is contrary to \( P = Q ; R \) where \( R \) is following the execution of \( Q \).
describe that the trace does not call any other process before $Q$. For example, in the Elevator specification, we get

$$\text{maxTraces (Work, main)} = \{\langle \text{stop, open} \rangle\},$$
$$\text{preTraces (main, close)} = \{\langle \text{req} \rangle\}.$$  

Now we introduce the predicate $wlp$ which describes a weakest liberal precondition and ensures that after the possible execution of a specific trace a certain predicate of the Object-Z part holds.\(^6\)

**Definition 5.** (weakest liberal precondition)

Let $p$ be a predicate, $m.i.o \in \text{Events}$, where $m \in \text{Methods}$ with input and output parameters $i$ and $o$ and $tr \in \text{Events}^*$. The predicate $wlp$ is defined inductively as

$$wlp(\langle \rangle, p) = p$$

$$wlp(\langle m.i.o \rangle, p) = \forall \text{State}' \bullet (\text{com}_{m}(i, o) \Rightarrow p')$$

$$wlp(\langle m.i.o \rangle \triangleright tr, p) = wlp(\langle m.i.o \rangle, wlp(tr, p))$$

For all $z \in \text{State}$ and $tr \in \text{Events}^*$, $z \models wlp(tr, p)$ iff every execution of $tr$ starting in $z$ leads to a state that satisfies $p$. Note that a primed predicate — as depicted in the second condition — refers to the states after an operation was executed.

The next lemma relates the definition of $wlp$ to the operational semantics of Object-Z:

**Lemma 1.** (Relation of $wlp$ to operational semantics)

For all $z \in \text{State}$, $tr \in \text{Events}^*$ and all predicates $p$ the following holds:

$$z \models wlp(tr, p) \iff \forall z' : \text{State}' \bullet (z \triangleright tr \Rightarrow_{\text{OZ}} z') \Rightarrow (z' \models p)$$

For both directions, the proof is based on an induction on the length of $tr$. See the appendix for details.

As a last preparation step we characterize the reachable states of a CSP-OZ specification $C$ as follows:

$$\text{reach}(C) := \{(z, P) \in \text{State}_C \mid \exists z_0 \in \text{State}, tr \in \text{Events}^* \bullet$$

$$z_0 \models \text{Init} \land (z_0, \text{main}) \triangleright tr (z, P)\}$$

The following theorem states our main result which we already motivated at the beginning of this section:

\(^6\) Contrary to total correctness specifications dealing with weakest preconditions, we use partial correctness here. Partial correctness does not require that a certain Object-Z operation terminates, e.g. that there always exists a successor state.
Theorem 1. (CLIs for CSP processes)
Let C be a CSP-OZ class such that its CSP part has the following structure:

\[
\begin{align*}
\text{main} & = \ldots \\
\text{P}_1 & = \ldots \\
& \vdots \\
\text{P}_n & = \ldots 
\end{align*}
\]

Let \( P_0 := \text{main} \). Let e \in \text{Events} and \( Q \in \{ P_0, \ldots, P_n \} \), be a process in which e occurs at a uniquely identified position. We need to find predicates CLI\(_i\) over State, \( i = 0..n \), such that the following three conditions are satisfied:

\[(\text{Init})\] \( \text{Init} \Rightarrow \text{CLI}_{\text{main}} \)
\[(\text{Precond})\] \( \forall \text{tr} \in \text{preTraces}(Q, e) \bullet \text{CLI}_Q \Rightarrow \text{wlp}(\text{tr}, \text{pre } e) \)
\[(\text{Cont})\] \( \forall j, k : 0..n; \forall \text{tr} \in \text{maxTraces}(P_j, P_k) \bullet \text{CLI}_{P_j} \Rightarrow \text{wlp}(\text{tr}, \text{CLI}_{P_k}) \)

Then the following holds for all \( z \in \text{State} \) and all CSP processes \( P \):

\[(z, P) \in \text{reach}(C) \land e \in \text{initials}(P) \Rightarrow e \in \text{init}(z)\]

The conditions guarantee that the precondition of event e is satisfied whenever the specification’s CSP part allows its execution, i.e., e never blocks.

Proof: Let \((z, P) \in \text{reach}(C)\) such that \( e \in \text{initials}(P) \). Since \( z \) is a reachable state in the Object-Z part, there exists \( z_0 \in \text{State} \) with

\[
(z_0, \text{main}) \xrightarrow{tr_1} (z, P)
\]

and \( z_0 \models \text{Init} \). We split the trace \( tr \), which leads to this state, into

\[
tr_{\text{rec}} \sim tr_{\text{fin}} \text{ such that } tr_{\text{rec}} = tr_1 \sim \cdots \sim tr_k,
\]

with \( tr_i \in \text{maxTraces}(P_{r_i}) \) and \( tr_{\text{fin}} \in \text{preTraces}(Q, e), r_i \in \{0, \ldots, n\} \). This split corresponds to a separation of \( tr \) into parts \( tr_i \) of the right sides of the corresponding process \( P_{r_i} \) between the recursive calls and the end piece \( tr_{\text{fin}} \) (possibly the empty trace). \( tr_{\text{fin}} \) will be executed after the last recursive call and leads to the specific occurrence of \( e \). Let \((z_i, P_{r_i})\) be the state after execution of the trace \( tr_1 \sim \cdots \sim tr_i \) with \( i \leq k \):

\[
(z_0, \text{main}) \xrightarrow{tr_i} (z_1, P_{r_1}) \xrightarrow{tr_2} \cdots \xrightarrow{tr_k} (z_k, Q) \xrightarrow{tr_{\text{fin}}} (z, P)
\]

\( (\text{Init}) \) yields \( z_0 \models \text{CLI}_{\text{main}} \). Additionally we get \( \text{main} \xrightarrow{tr_1} \text{CSP } P_{r_1} \). The fact that \( tr_1 \in \text{maxTraces}(\text{main}) \) allows us to apply \( (\text{Cont}) \) to deduce \( z_0 \models \text{wlp}(tr_1, \text{CLI}_{P_{r_1}}) \). By definition of \( \text{wlp} \) and because of

\[
(z_0, \text{main}) \xrightarrow{tr_i} (z_1, P_{r_1}),
\]
we get $z_1 \models \text{CLI}_{P_1}$. This procedure will be repeated for every $tr_i$ until we reach $z_k \models \text{CLI}_{P_k}$. $tr_{\text{fin}} \in \text{preTraces}(Q, e)$ allows us to apply (Precond) and we get $z_k \models wlp(tr_{\text{fin}}, \text{pre } e)$, i.e., $z \models \text{pre } e$ by applying Lemma 1. Finally by definition of init and pre this means $e \in \text{init}(z)$.

We illustrate Theorem 1 in connection with Lemma 1 by showing how to apply them to a set of process equations. Applied to an event $e$ occurring in a process $Q$, the first condition (Init) states that the CLI for main holds initially:

$$
\text{CLI}_0 \rightarrow \\
\begin{array}{l}
\text{main } = \ldots \rightarrow P_i \\
\ldots \\
Q = \ldots \rightarrow e \\
\ldots \\
P_j = \ldots \rightarrow P_k
\end{array}
$$

The second condition (Precond) describes that after executing any trace of $Q$ leading to $e$, its precondition is always satisfied if CLI$_Q$ holds prior to the beginning of the trace. Note, that we apply Lemma 1 here since we no longer use the term $wlp(\ldots)$ but instead unfold the trace which leads to the predicate pre $e$.

$$
\text{CLI}_Q \rightarrow \\
\begin{array}{l}
\text{main } = \ldots \rightarrow P_i \\
\ldots \\
Q \xrightarrow{tr_{\text{fin}}} = \ldots \rightarrow e \\
\ldots \\
P_j = \ldots \rightarrow P_k
\end{array} \leftarrow \text{pre } e
$$

Finally, condition (Cont) guarantees that after execution of a trace leading from one process to the next, the associated CLI holds if the CLI for the first process holds prior to the beginning of the trace. Again we apply Lemma 1 to illustrate this:

$$
\text{CLI}_{P_j} \rightarrow \\
\begin{array}{l}
\text{main } = \ldots \rightarrow P_i \\
\ldots \\
Q = \ldots \rightarrow e \\
\ldots \\
P_j \xrightarrow{tr} = \ldots \rightarrow P_k
\end{array} \leftarrow \text{CLI}_{P_k}
$$

### 4.2 Results applied to Elevator

We again take a look at our example specification of an elevator and apply Theorem 1. Similar to an invariant search in program verification, we need to find a set of CLIs that satisfy the conditions of Theorem 1. For event
req in the Elevator specification the following conditions for the CLIs must hold:

\begin{align}
\text{(Init)} & \quad \text{Init} \Rightarrow \text{CLI}_{\text{main}} \\
\text{(Precond)} & \quad \text{CLI}_{\text{main}} \Rightarrow \text{wlp}(⟨⟩, (aFloor = rFloor)) \\
\text{(Cont)} & \quad \\
& \quad \text{CLI}_{\text{main}} \Rightarrow \text{wlp}(⟨\text{req}, \text{close}⟩, \text{CLI}_{\text{Work}}) \\
& \quad \text{CLI}_{\text{Work}} \Rightarrow \text{wlp}(⟨\text{move}⟩, \text{CLI}_{\text{Work}}) \\
& \quad \text{CLI}_{\text{Work}} \Rightarrow \text{wlp}(⟨\text{stop}, \text{open}⟩, \text{CLI}_{\text{main}})
\end{align}

\text{CLI}_{\text{main}} needs to hold initially according to \text{(Init)}. Additionally, the event \text{req} is the first event on the right hand side of \text{main}, i.e., for condition \text{(Precond)} we get \text{preTraces}(\text{main}, e) = \{⟨⟩\}. That is why we only need to consider one implication for condition \text{(Precond)}. Finally, \text{(Cont)} identifies three conditions on maximal traces: for each of the three possible branches in the CSP part there exists exactly one maximal trace. A solution for this set of implications are two invariants \text{CLI}_{\text{main}} and \text{CLI}_{\text{Work}} resolving all implications conjointly.

A basic idea to find such a solution is to write down each single condition as in (2) and in a next step to approximate several predicates. To this end, we use the condition of the right hand side of \text{(Precond)} for \text{CLI}_{\text{main}}. In addition, this predicate has to be weak enough to be implied by \text{Init} and also by an appropriate \text{CLI}_{\text{Work}}, added to the precondition and effect of the trace \text{tr} = ⟨\text{stop}, \text{open}⟩. \text{tr} = ⟨\text{stop}, \text{open}⟩ can only be executed if \text{aFloor} = \text{rFloor} holds. Since this property is not changed by the effect of \text{tr}, the third implication of \text{(Cont)} holds without adding a nontrivial CLI for process \text{Work}. Thus defining \text{CLI}_{\text{Work}} ≡ \text{true} is enough since \text{CLI}_{\text{main}} does not appear on the right hand side of the other implications of condition \text{(Cont)}. Therefore one possible solution for all conditions is the following set of CLIs:

\begin{align}
\text{CLI}_{\text{main}} & \equiv (\text{aFloor} = \text{rFloor}) \\
\text{CLI}_{\text{Work}} & \equiv \text{true}
\end{align}

These CLIs thus ensure that \text{req} is enabled at any execution allowed by the specification’s CSP part.

For events \text{close} and \text{open} we can similarly find CLIs satisfying the conditions. For event \text{move} we need to find CLIs that — amongst others — satisfy these conditions:

\begin{align}
\text{(Precond)} & \quad \text{CLI}_{\text{Work}} \Rightarrow \text{wlp}(⟨⟩, (aFloor \neq rFloor)) \\
\text{(Cont)} & \quad \text{CLI}_{\text{Work}} \Rightarrow \text{wlp}(⟨\text{move}⟩, \text{CLI}_{\text{Work}})
\end{align}

On the one hand, we need to deduce \text{CLI}_{\text{Work}} ⇒ aFloor \neq rFloor. On the other hand, an execution of \text{move} may change the relation \text{aFloor} \neq \text{rFloor} to \text{aFloor} = \text{rFloor}. Therefore, no solution can be found in this case: the
solution must imply \( \text{CLI}_{\text{Work}} = a\text{Floor} \neq r\text{Floor} \) because of condition (Pre-cond) but (Cont) makes it impossible to add this predicate to a possible solution. The same holds for stop.

Summarizing, it can be stated that we established a way to prove that certain Object-Z operations are always enabled when control flow of the CSP part reaches them. In our example, this holds for events req, close and open even though these events have a non-trivial precondition. In the next section we will exploit this additional knowledge to improve the SDG and show how this modification affects the outcome of our slicing approach.

5. Combination: Improved Slicing

The results that we obtained in the previous section from Theorem 1 lead us directly to our goal, namely to optimize our slicing approach.

This can easily be seen in the elevator example: from the fact that req never blocks we infer that there exists no real control dependence originating from req, since in spite of its non-trivial precondition req does not determine whether control flow reaches the subsequent node close or not. Therefore we can remove the associated control dependence edge req \( \rightarrow \) close from the original SDG and replace it by an edge start.main \( \rightarrow \) close from req’s predecessor start.main to close such that the refined SDG remains well-defined. The associated control dependence edges for close and open can also be removed from the SDG, while the edges associated to move and stop cannot be removed since we were not able to find suitable CLIs for these events.

All in all, these optimizations yield a significantly reduced SDG as can be seen in Figure 3. The main difference in comparison to the previous SDG in Figure 2 are the omitted control dependence edges between nodes inside the newly introduced boxes. These boxes collect nodes which are not control dependent on each other. The omitted edges have been replaced by a single control dependence edge between the first predecessor node outside the box and the box itself. This single edge is a graphical abbreviation for multiple control dependence edges between its source node and each node inside the box.

To illustrate the effect of this modification on the slicing outcome, we compute the slice for property \( \square \diamond (a\text{Floor} = r\text{Floor}) \), that describes the fact that the elevator will always eventually reach its target floor.

The slice computation starts with the set of nodes \{req, move\} that directly influence the given property by modifying variables rFloor and aFloor mentioned in the formula. When we compute the set of SDG nodes that is backwards reachable from the initial set of nodes, we do now no longer reach all nodes, since open and close can not be reached via control or data dependencies: starting for instance from req, we can reach node call.main
via start.main, but the next node we reach is stop, since open does no longer have a control dependence edge leading to call.main.

All in all, our improved slicing algorithm can achieve reductions in cases where previously this was not possible. In comparison to the original specification, its slice with respect to $\Box \Diamond (aFloor = rFloor)$ does not contain the schemas com_open and com_close as well as variable door. This result is sound, since opening and closing the elevator’s door are aspects of the original specification that do not have any influence on the given property.

5.1 Correctness of Improved Slicing

In [1], we gave the correctness proof for our original slicing approach. We will explain now, why this proof still holds for our improved slicing approach.

The only modification that we have introduced here to the specification dependence graph is the removal of certain control dependence edges. A similar kind of modification takes already place during the ordinary construction of the specification dependence graph: if an event has a trivial precondition (i.e., a precondition equivalent to true), its associated node is not source of any control dependence edge. In this paper we have shown that an event with non-trivial preconditions (i.e., a precondition not equivalent to true) can nevertheless be regarded as an event with trivial preconditions, provided that we can find CLIs for this event that satisfy the conditions as given in Theorem 1. In this case, the event’s precondition is not always equivalent to true, but at any execution of the event that is allowed by the specification’s CSP part. Consequently, we can treat such an event in the same way as an event with trivial preconditions which is exactly what our
approach does in the construction of the specification dependence graph.

Since our proof in [1] does already cover events with trivial preconditions, it does also cover such additional events with non-trivial preconditions without any further changes, provided that Theorem 1 is applicable to these events.

6. Conclusion

In this paper we have proposed the combination of the (syntax-based) construction of a dependence graph with small deductive proof steps. The results of these proofs can be used to eliminate dependencies, thus to optimize slicing and improve the runtime of the successive model checking step.

Related work. Slicing of formal specifications has mainly been done for Z specifications [4, 22]. These approaches, however, do not consider verification, i.e., slicing is not carried out with respect to temporal logic properties of the specification. Work on slicing used for reducing programs before verification has for instance been done in [8] for Java (preserving LTL_{−X} properties) and in [20] for SAL programs (preserving CTL^∗_{−X} properties). The approach closest to ours is [17]. It proposes a combination of program slicing (of C programs) with constraint solving with a similar goal: elimination of some data dependencies. In contrast to our approach, however, slicing remains in its original domain of program analysis and is not yet considered to be applied in the context of formal verification.

As future work we intend to investigate (in collaboration with our project partners of the University of Saarland) how to automatically compute the enabledness of operations, so that the deductive manual proof steps can be replaced by automatic procedures.

Acknowledgements

This research was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS, for further information see www.avacs.org).

References

Appendix

Proof of Lemma 1: Let $z \in \text{State}, tr \in \text{Events}^*, p \in \text{ZPred}$. 

$\Rightarrow$ (Correctness):

Let $z \models \text{wlp}(tr, p)$. We use induction on the length of $tr$:

Induction base:
\( tr = \langle \rangle \) is obvious since \( wlp(\langle \rangle, p) = p \).

\( tr = \langle m.i.o \rangle \) with \( m \in \text{Methods} \): if \( z \overset{tr}{\Rightarrow}_O z' \), \( z \models \text{com}_m(i, o) \) holds. By definition of \( wlp \) we deduce \( z' \models p \).

**Induction hypothesis:**
Let the proposition be true for all \( tr \) with \( \#tr < n \).

**Induction step:**
Let \( tr = \langle m.i.o \rangle \prec tr' \) for \( m \in \text{Methods} \) and \( tr' \in \text{Events}^* \) with \( \#tr = n \). By definition of \( wlp \) we deduce:

\[
\begin{align*}
\text{wlp}(tr, p) &= \text{wlp}(\langle m.i.o \rangle, \text{wlp}(tr', p)) \\
&= \forall \text{State}' \bullet (\text{com}_m(i, o)) \Rightarrow \text{wlp}'(tr', p)
\end{align*}
\]

(2)

To prove our statement, let \( \hat{z}, z' \in \text{State}' \) with \( z \overset{\langle m.i.o \rangle}{\Rightarrow}_O \hat{z} \overset{tr'}{\Rightarrow}_O z' \). In particular, \( z \models \text{com}_m(i, o) \) which is the assumption for (2). From this we deduce \( \hat{z} \models \text{wlp}(tr', p) \). We apply the induction hypothesis and because of \( \hat{z} \overset{tr'}{\Rightarrow}_O z' \) we get \( z' \models p \).

\( \Rightarrow (\text{Completeness}) \):
Let \( z \models \forall z' : \text{State}' \bullet (z \overset{tr}{\Rightarrow}_O z') \Rightarrow (z' \models p) \). We have to show that \( z \models \text{wlp}(tr, p) \) and again prove this by induction on the length of \( tr \):

**Induction base:**
\( tr = \langle \rangle \) is obvious.

\( tr = \langle m.i.o \rangle \) with \( m \in \text{Methods} \): if \( z \models \text{com}_m(i, o) \) holds, by the operational semantics we get \( z \overset{\langle m.i.o \rangle}{\Rightarrow}_O z' \). This yields that every \( m.i.o \)-successor of \( z \) satisfies \( p \).

**Induction hypothesis:**
Let the proposition be true for all \( tr \) with \( \#tr < n \).

**Induction step:**
Again let \( tr = \langle m.i.o \rangle \prec tr' \) with \( \#tr = n \). We assume

\[
\forall z' : \text{State}' \bullet (z \overset{tr}{\Rightarrow}_O z') \Rightarrow (z' \models p)
\]

(3)

and by definition of \( wlp \) have to show that \( z \models \text{wlp}(\langle m.i.o \rangle, \text{wlp}(tr', p)) \), which means

\[
z \models (\forall \text{State}' \bullet (\text{com}_m(i, o)) \Rightarrow \text{wlp}'(tr', p))
\]

This means that for every \( m.i.o \)-successor \( \hat{z} \) of \( z \text{wlp}(tr', p) \) holds. By induction hypothesis applied to \( tr' \) it is sufficient to show that the following
holds:
\[ \forall z' : \text{State}' \cdot (\hat{z} \xrightarrow{\text{tr}' \ OZ} z') \Rightarrow (z' \models p') \]

We take a look at the following picture:

We know that \( z \langle \text{m.i.o} \rangle \models OZ \hat{z} \) and \( \hat{z} \xrightarrow{\text{tr}' \ OZ} z' \) holds. This yields \( z \langle \text{m.i.o} \rangle \xrightarrow{\text{tr}' \ OZ} z' \) and with (3) we deduce \( z' \models p \). This completes the proof. \( \square \)