Nordic Journal of Computing 13(2006), 22–45.

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
HEIKE WEHRHEIM
Universität Paderborn, Institut für Informatik
33098 Paderborn, Germany
{bmetzler|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. During 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.

ACM CCS Categories and Subject Descriptors: D.2 [Software Engineering]; D.2.1 [Software Engineering]: Requirements/Specification; D.2.4 [Software Engineering]: Software/Program Verification

Key words: program slicing, integrated formal specifications, exact reduction, deductive verification, model checking

1. Introduction

Slicing (see [46]) originates from the area of program analysis [37] where it has first been employed for debugging programs. The initial idea of Weiser [52] 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 [18].

Received January 27, 2006; revised May 18, 2006; accepted June 14, 2006.

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 [10], partial-order reduction [42], heuristic search [12] etc.).

Slicing usually starts with the construction of a dependence graph [24] 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 [35].

In this paper, we are concerned with slicing of specifications written in an integrated notation (called CSP-OZ [13]), combining two well known formal specification techniques: CSP [22] for *behavioural aspects* of systems and Object-Z (OZ) [44] 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 [23]) 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 [5] and [4]. 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 examples studied 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 irrelevant 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 [48]. 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 Fig. 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 `main` = ...), the passenger requests to be delivered to a certain floor (event *req*, connected by the CSP *action prefix* operator $\rightarrow$ to the remainder of the `main` process), upon which the door *close*s (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 *move*s, followed by another call of process *Work*, or it *stop*s, followed by *open*ing its doors and returning to the initial `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 consists of two parts: the upper part may contain a Δ-list of variables that are modified in the lower part. In schema `com_`*move* for instance, Δ(*aFloor*) indicates that variable *aFloor* is modified when event *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. $n? : \mathbb{N}$ in schema `com_`*req* indicates that the elevator's environment will provide an input $n?$ of type $\mathbb{N}$ when event *req* takes place). The lower part can contain a set of predicates over primed and unprimed variables and over input

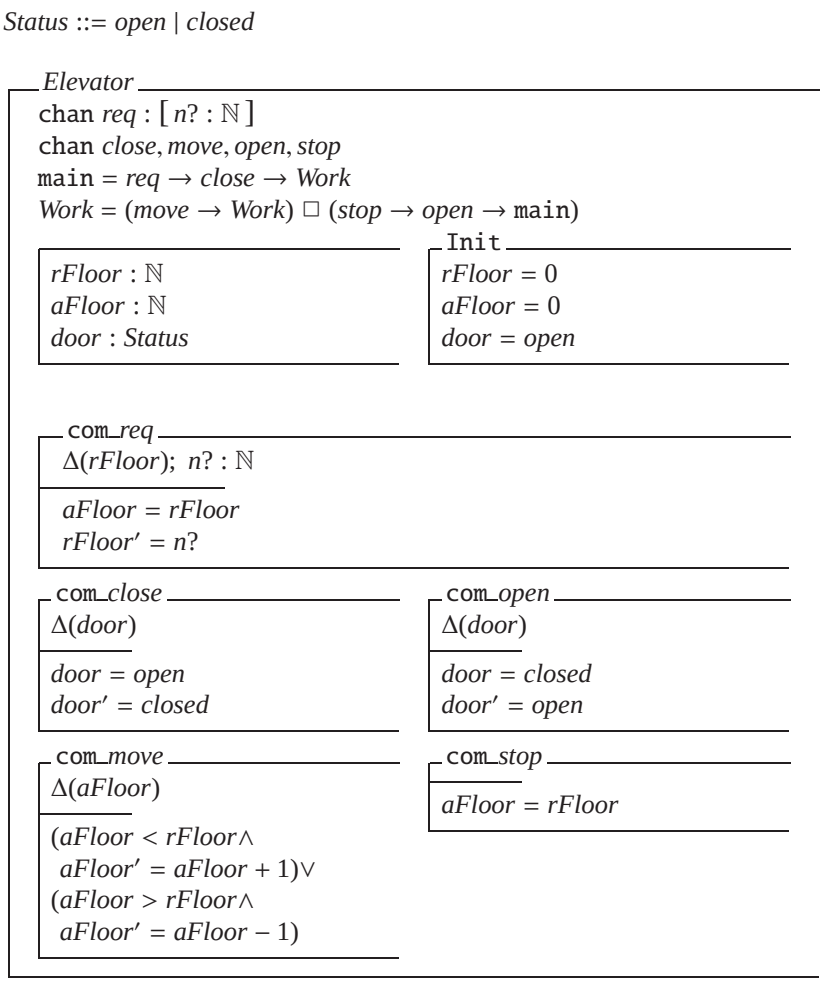


Fig. 1: Elevator specification.

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 *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 pre [53]:

$$\mathrm{pre}\, Op = \exists\, State', outputs \bullet Op$$

The predicate is true if an after state (*State′*) and an output (*outputs*) can be found which make the after predicate in schema *Op* true, i.e., the application of the pre operator to a schema *Op* yields exactly the predicate that forms the schema's precondition. When this predicate is unsatisfied, its associated event is blocked, otherwise it is enabled. In schema com_*open*, for instance, predicate *door* = *closed* defines a precondition that blocks event *open*, unless the *door* is *close*d.

The *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 `com_open`, for instance, predicate $door' = open$ defines *door* to be *open* after event *open* has taken place.

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.

D 1. (L K) *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 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* $L : S \rightarrow 2^{AP}$.

Atomic propositions are equalities of elements of the state with constant values, such as $door = open$ or $aFloor = 0$ for our example. 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 $Z = (State, \texttt{Init}, (\texttt{com\_}m)_{m \in Methods})$ be the Object-Z part of a CSP-OZ class C with *State* referring to the class' state schema, `Init` to its initial state schema and *Methods* to the set of all methods used in the Object-Z part. Let `main` be the CSP part of a CSP-OZ class C. The alphabet that we use here is the set $Events = \{m.i.o \mid m \in Methods, i \in in(m), o \in out(m)\}$ 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 $in(m)$ and $out(m)$ respectively being the set of all input and output parameters of m. For sake of simplicity we restrict methods to having only single input and output parameters. This can easily be extended to the general case of sets of input and output parameters. We assume the CSP part to be *data independent* in that it neither restricts nor uses values of parameters.

D 2. (K O -Z) *The Kripke structure semantics of the Object-Z part is defined as the labelled Kripke structure*

$$K^{OZ} = (State, Init, \rightarrow_{OZ}, L^{OZ})$$

with the labelling function L^{OZ} *mapping each state onto the set of atomic propositions over the Object-Z state space that are valid in this state, Init being the set*

of states that satisfy the Object-Z part's `Init` *schema and the transition relation* $\rightarrow_{OZ} = \{(z, m.i.o, z') \mid \texttt{com\_}m(z, i, o, z')\}$ *relating pre (z) and post (z′) states according to the definition of associated Object-Z methods.*

Note that we use *blocking semantics* here: the precondition of an operation (pre e) acts as a guard for its execution. Instead of $(z, e, z') \in \rightarrow_{OZ}$ we also write $z \overset{e}{\rightarrow}_{OZ} z'$ and instead of $z \overset{e_1}{\rightarrow} \ldots \overset{e_n}{\rightarrow} z'$ we also write $z \overset{\langle e_1,\ldots,e_n\rangle}{\Rightarrow}_{OZ} z'$. For any $z \in State$ we furthermore define:

- $z \overset{e}{\rightarrow}_{OZ} :\Leftrightarrow \exists z' \in State \bullet z \overset{e}{\rightarrow}_{OZ} z'$,
- $z \overset{tr}{\Rightarrow}_{OZ} :\Leftrightarrow \exists z' \in State \bullet z \overset{tr}{\Rightarrow}_{OZ} z'$,
- $init(z) := \{e \in Events \mid z \overset{e}{\rightarrow}_{OZ}\}$.

D 3. (K CSP) *The Kripke structure semantics of the CSP part is defined as the labelled Kripke structure*

$$K^{CSP} = (CSP, \{\texttt{main}\}, \rightarrow_{CSP}, L^{CSP})$$

with CSP denoting the set of all CSP terms, `main` *being the only initial CSP term,* $\rightarrow_{CSP}$ *being the transition relation defined according to the operational semantics of CSP [43] and with the labelling function* L^{CSP} *mapping each CSP process onto the set of all atomic propositions over the Object-Z state space.*

The labelling function for the CSP part's Kripke structure does not impose any restrictions with respect to the Object-Z state space since the CSP part makes no restrictions on values of attributes of the class.

D 4. (K CSP-OZ) *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:* $K = (State_C, Init_C, \rightarrow, L_C)$ *with* $State_C = State \times CSP$, $Init_C = Init \times \{\texttt{main}\}$, $L_C(z, P) = L^{OZ}(z) \cap L^{CSP}(P)$ *and*

$$\rightarrow = \{((z, P), e, (z', P')) \mid (e \neq \tau, P \overset{e}{\rightarrow}_{CSP} P', z \overset{e}{\rightarrow}_{OZ} z') \vee (e = \tau, P \overset{\tau}{\rightarrow}_{CSP} P', z = z')\}$$

The relation $\Rightarrow$ for traces is defined accordingly. The special symbol τ describes an internal event of the CSP process. It is not observable in any trace and not part of the set *Events*.

For describing properties of CSP-OZ classes we can now use any stuttering invariant[1] temporal logic which can be interpreted on labelled Kripke structures.

[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.

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 φ holds on the Kripke structure of the specification C and C_{red} denotes the reduced specification):

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

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 [4] 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. In the simple case of our elevator example the CFG is equivalent to its SDG (which will be defined next) without data dependence edges (see Fig. 2).

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 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.

[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.

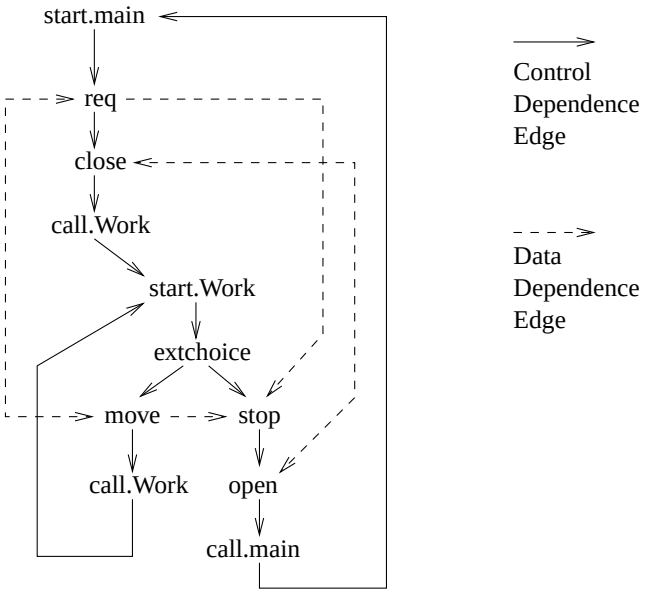


Fig. 2: Elevator SDG.

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 Fig. 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 [4].

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`:

$$\begin{array}{lcl} P_0 & = & \ldots \quad \rightarrow P_{i_1} \\ P_1 & = & \ldots \quad \rightarrow P_{i_2} \\ & \vdots & \\ P_n & = & \ldots \quad \rightarrow P_{i_n} \end{array}$$

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 5):

$$P_0 \stackrel{tr}{\Rightarrow}_{CSP} P' \text{ with } P' \stackrel{e}{\rightarrow}_{CSP} P''$$

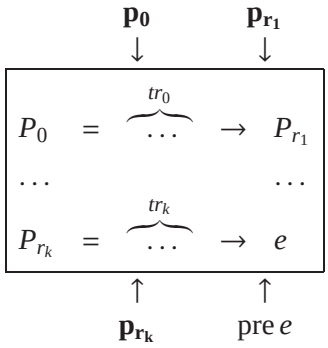
can be viewed as

$$P_0 \stackrel{tr_0}{\Rightarrow}_{CSP} P_{r_1} \stackrel{tr_1}{\Rightarrow}_{CSP} \cdots \stackrel{tr_{k-1}}{\Rightarrow}_{CSP} P_{r_k} \stackrel{tr_k}{\Rightarrow}_{CSP} P' \tag{1}$$

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

[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.

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 pre *e* must hold:



Our approach uses deductive verification techniques and is similar to the idea that [48] applied to prove deadlock freedom of CSP || 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.

4.1 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 [7]. 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]

[4] The term *control* indicates that the CSP part controls the order of execution of the Object-Z schemas.

[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.

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 $PId \cap Methods = \varnothing$. The mapping $traces(P)$ computes the set of all possible traces of a CSP process P. Finally, the set

$$initials(P) = \{e \in Events \mid \langle e \rangle \in traces(P)\}$$

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

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:

D 5. *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*

$$maxTraces\ (P, Q) = \{tr \in traces\ (P[\texttt{STOP}/PId]) \mid P \overset{tr}{\Rightarrow}_{CSP} Q\}$$

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

$$preTraces\ (P, e) = \{tr \in traces\ (P[\texttt{STOP}/PId]) \mid \exists Q \bullet P \overset{tr}{\Rightarrow}_{CSP} Q \wedge e \in initials(Q)\}$$

$P[\texttt{STOP}/PId]$ describes the process P with every occurrence of a process identifier replaced by STOP. By using process replacement, we are able to describe that the trace does not call any other process before Q. For example, in the *Elevator* specification, we get

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

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]

D 6. ( ) *Let p be a predicate, m.i.o ∈ Events, where m ∈ Methods with input and output parameters i and o and tr ∈ Events*. The predicate* wlp *is defined inductively as*

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

[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.

For all $z \in State$ and $tr \in Events^*$, $z \models wlp(tr, p)$ 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:

L 1. (R *wlp*) *For all* $z \in State$, $tr \in Events^*$ *and all predicates p the following holds:*

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

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 class C as follows:

$$reach(C) := \{(z, P) \in State_C \mid \exists z_0 \in State, tr \in Events^* \bullet z_0 \models \texttt{Init} \wedge (z_0, \texttt{main}) \overset{tr}{\Rightarrow} (z, P)\}$$

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

T 1. (CLI CSP) *Let C be a CSP-OZ class such that its CSP part has the following structure:*

$$\begin{array}{ll} \texttt{main} & = \ldots \\ P_1 & = \ldots \\ & \vdots \\ P_n & = \ldots \end{array}$$

Let $P_0 := \texttt{main}$. *Let* $e \in Events$ *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_{P_i} *over State,* $i = 0..n$, *such that the following three conditions are satisfied:*

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

Then the following holds for all $z \in State$ *and all CSP processes P:*

$$(z, P) \in reach(C) \wedge e \in initials(P) \Rightarrow e \in init(z)$$

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.

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

$$(z_0, \texttt{main}) \overset{tr}{\Rightarrow} (z, P)$$

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

$$tr_{rec} \frown tr_{fin} \text{ such that } tr_{rec} = tr_1 \frown \cdots \frown tr_k,$$

with $tr_i \in \mathit{maxTraces}(P_{r_i})$ and $tr_{fin} \in \mathit{preTraces}(Q, e)$, $r_i \in \{0, \ldots, n\}$. 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_{fin} (possibly the empty trace). tr_{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 $tr_1 \frown \cdots \frown tr_i$ with $i \leq k$:

$$(z_0, \texttt{main}) \overset{tr_1}{\Rightarrow} (z_1, P_{r_1}) \overset{tr_2}{\Rightarrow} \cdots \overset{tr_k}{\Rightarrow} (z_k, Q) \overset{tr_{fin}}{\Rightarrow} (z, P)$$

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

$$(z_0, \texttt{main}) \overset{tr_1}{\Rightarrow} (z_1, P_{r_1}),$$

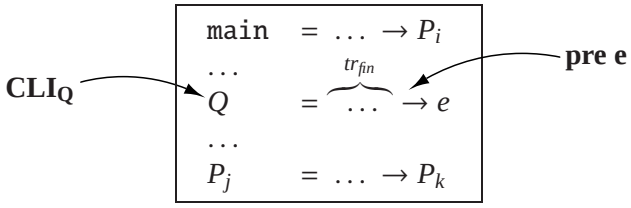
we get $z_1 \models \mathrm{CLI}_{P_{r_1}}$. This procedure will be repeated for every tr_i until we reach $z_k \models \mathrm{CLI}_{P_{r_k}}$. $tr_{fin} \in \mathit{preTraces}(Q, e)$ allows us to apply **(Precond)** and we get $z_k \models \mathit{wlp}(tr_{fin}, \mathrm{pre}\, e)$, i.e., $z \models \mathrm{pre}\, e$ by applying Lemma 4.1. Finally by definition of *init* and pre this means $e \in \mathit{init}(z)$. □

We illustrate Theorem 1 in connection with Lemma 4.1 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:

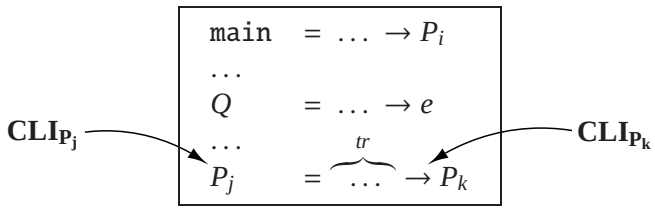
CLI₀

`main`	= ...	$\rightarrow P_i$
...		
Q	= ...	$\rightarrow e$
...		
P_j	= ...	$\rightarrow P_k$

The second condition **(Precond)** describes that after executing any trace of Q leading to e, its precondition is always satisfied at this specific occurrence of e if CLI_Q holds prior to the beginning of the trace. Note, that we apply Lemma 4.1 here since we no longer use the term $\mathit{wlp}(\ldots)$ but instead unfold the trace which leads to the predicate $\mathrm{pre}\, e$.



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 4.1 to illustrate this:



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:

$$\begin{aligned}
\textbf{(Init)}\ & \texttt{Init} \Rightarrow \mathrm{CLI}_{\texttt{main}} \\
\textbf{(Precond)}\ & \mathrm{CLI}_{\texttt{main}} \Rightarrow \mathit{wlp}(\langle\,\rangle, (\mathit{aFloor} = \mathit{rFloor})) \\
\textbf{(Cont)}\ & \mathrm{CLI}_{\texttt{main}} \Rightarrow \mathit{wlp}(\langle \mathit{req}, \mathit{close}\rangle, \mathrm{CLI}_{\mathit{Work}}) \\
& \mathrm{CLI}_{\mathit{Work}} \Rightarrow \mathit{wlp}(\langle \mathit{move}\rangle, \mathrm{CLI}_{\mathit{Work}}) \\
& \mathrm{CLI}_{\mathit{Work}} \Rightarrow \mathit{wlp}(\langle \mathit{stop}, \mathit{open}\rangle, \mathrm{CLI}_{\texttt{main}})
\end{aligned} \tag{2}$$

$\mathrm{CLI}_{\texttt{main}}$ needs to hold initially according to **(Init)**. Additionally, the event *req* is the first event on the right hand side of `main`, i.e., for condition **(Precond)** we get $\mathit{preTraces}(\texttt{main}, e) = \{\langle\,\rangle\}$. That is why we only need to consider one implication for condition **(Precond)**. Finally, **(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 $\mathrm{CLI}_{\texttt{main}}$ and $\mathrm{CLI}_{\mathit{Work}}$ 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 **(Precond)** for $\mathrm{CLI}_{\texttt{main}}$. In addition, this predicate has to be weak enough to be implied by `Init` and also by an appropriate $\mathrm{CLI}_{\mathit{Work}}$, added to the precondition and effect of the trace $tr = \langle \mathit{stop}, \mathit{open}\rangle$. $tr = \langle \mathit{stop}, \mathit{open}\rangle$

can only be executed if $aFloor = rFloor$ holds. Since this property is not changed by the effect of *tr*, the third implication of **(Cont)** holds without adding a nontrivial CLI for process *Work*. Thus defining $\mathrm{CLI}_{Work} \equiv \text{true}$ is enough since $\mathrm{CLI}_{\texttt{main}}$ does not appear on the right hand side of the other implications of condition **(Cont)**. Therefore one possible solution for all conditions is the following set of CLIs[7]

$$\mathrm{CLI}_{\texttt{main}} \equiv (aFloor = rFloor)$$
$$\mathrm{CLI}_{Work} \equiv \text{true}$$

These CLIs thus ensure that *req* is enabled at any execution allowed by the specification's CSP part.

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

$$\textbf{(Precond)}\ \mathrm{CLI}_{Work} \Rightarrow wlp(\langle\,\rangle, (aFloor \neq rFloor))$$
$$\textbf{(Cont)}\ \mathrm{CLI}_{Work} \Rightarrow wlp(\langle move\rangle, \mathrm{CLI}_{Work})$$

On the one hand, we need to deduce $\mathrm{CLI}_{Work} \Rightarrow aFloor \neq rFloor$. On the other hand, an execution of *move* may change the relation $aFloor \neq rFloor$ to $aFloor = rFloor$. Therefore, no solution can be found in this case: the solution must imply $\mathrm{CLI}_{Work} = aFloor \neq rFloor$ because of condition **(Precond)** 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 each of these events has 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

[7] Note that this solution is not unique, since, for instance, $\mathrm{CLI}_{\texttt{main}}$ can be arbitrarily strengthened with predicates satisfying the `Init` schema. This yields numerous different solutions, whereas the given solution is the weakest possible.

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 Fig. 3. The main difference in comparison to the previous SDG in Fig. 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 $\Box\Diamond(aFloor = rFloor)$, 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*.

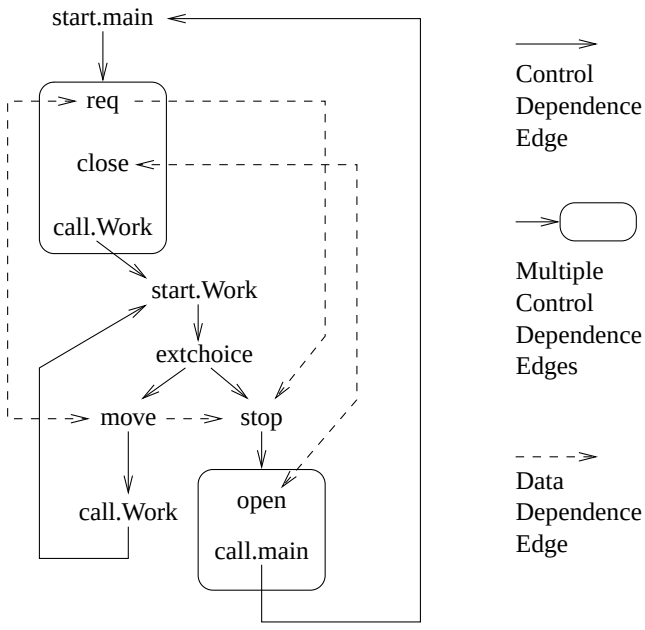


Fig. 3: Elevator SDG.

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 [4], 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 [4] 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 subsequent steps of model generation and model checking.

Related work. *Program slicing* as originally defined by Weiser [51] has been enhanced and modified multiple times with respect to many different aspects such as the approaches of program dependence graph based slicing [40], interprocedural slicing [24], dynamic slicing [1] or conditioned slicing [16]. At the same time it has found numerous fields of application (for overview papers see [46, 28, 17, 31, 55]) ranging from its traditional field of program analysis and debugging [52] over software maintenance [14] and reverse engineering [2, 25] up to model generation [18] and model checking [33, 11]. This goes along with a similarly wide spectrum of slicing targets including widely established programming languages such as C [47, 29] and Java [36, 18], model checker input languages such as

Promela [33] and SAL [15] and specification formalisms such as hierarchical state machines [19], WSL [50], Petri Nets [30] or – more frequently – Z specifications [39, 8, 54, 3] as in our case.

Formal verification is an application area of slicing that has recently seen increasing interest, since slicing seems to be one technique that can help to tackle the problem of state space explosion during model checking. Empirical results in this setting [26, 11] have shown that slicing can indeed effectively complement different other strategies such as compositional verification [20], efficient model representations [6], data abstraction [27], partial order reduction [42] and cone of influence reduction [9] that are mostly applied on a different stage than slicing, namely either during or after model generation has already been performed. In contrast to that, slicing can – as a syntax-based technique – be applied beforehand as a relatively cheap method to already reduce the input to model generation. Thus, the benefit of slicing can be seen in two directions: First, it enables an optimization by accelerating the process of model generation, which is for larger systems already a substantial part of the complete verification process. Second, it yields smaller models to which subsequently the mentioned orthogonal strategies for state space reduction can still be applied.

The existing approaches to static slicing of formal specifications, 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 [18] and [11] for Java (preserving LTL_{-X} properties) and in [15] for SAL programs (preserving CTL^{*}_{-X} properties).

The approach closest to ours is [45]. 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.

Deductive verification of software and hardware systems has its origin in early works by Hoare [21] who introduced the concept of invariants and proof rules for formal verification (Hoare logic). It is for instance the basis for temporal verification of reactive systems [32]. Deductive verification uses tools for theorem proving such as PVS [41] or Isabelle [38] (in contrast to automatic verification using model checking techniques). Our approach is motivated by a work of Treharne and Schneider [48] who coupled the B Method with CSP and used verification techniques based on weakest precondition semantics for CSP [34]. A more detailed description of their concept of Control Loop Invariants can be found in [49]. A weakest precondition semantics for Object-Z can for instance be found in [7].

Future work. 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. For preliminary experiments in this direction we applied backwards propagation of the desired enabledness property through the

control flow graph in order to show the property's inductiveness. This was successful in some cases, while for other (non-inductive) cases backwards propagation is not sufficient, but weakening becomes necessary, which seems to be difficult to automate and which can currently only manually be conducted.

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

[1] A , H. H , J. R. 1990. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 246–256.

[2] B , J. E , D. 1993. Program and interface slicing for reverse engineering. In *ICSE '93: Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 509–518.

[3] B , A. 2004. *Specification Comprehension — Reducing the Complexity of Specifications*. PhD thesis, Institute for Informatics-Systems, University of Klagenfurt.

[4] B ¨ , I. W , H. 2005. Slicing an Integrated Formal Method for Verification. In *ICFEM 2005: Seventh International Conference on Formal Engineering Methods*, Volume 3785 of *LNCS*. Springer, 360–374.

[5] B ¨ , I. W , H. 2005. Slicing Object-Z Specifications for Verification. In *ZB 2005*, Volume 3455 of *LNCS*. Springer, 414–433.

[6] B , R. E. 1992. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Comput. Surv. 24*, 3, 293–318.

[7] C , A. W , J. 1998. A Weakest Precondition Semantics for Z. *The Computer Journal 41*, 1, 1–15.

[8] C , D. R , D. 1994. Static and Dynamic Specification Slicing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 138–153.

[9] C , E., G , O., P , D. 1999. *Model checking*. MIT Press.

[10] C , E. M., G , O., J , S., L , Y., V , H. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, 154–169.

[11] D , M. B., H , J., H , M., R , V., W , R., W , T. 2006. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2006*.

[12] E , S., L , S., L -L , A. 2004. Directed Explicit-State Model-Checking in the Validation of Communication Protocols. *International Journal on Software Tools for Technology Transfer 5*, 2-3, 247–267.

[13] F , C. 1997. CSP-OZ: A Combination of Object-Z and CSP. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'97)*, Volume 2. Chapman & Hall, 423–438.

[14] G , K. B. L , J. R. 1991. Using Program Slicing in Software Maintenance. *IEEE Trans. Softw. Eng. 17*, 8, 751–761.

[15] G , V., S , H., S N. 1999. Slicing SAL. Tech. report, SRI International, http://theory.stanford.edu/.

[16] H , M., H , R. M., F , C., D , S., H , J. 2001. Pre/Post Conditioned Slicing. In *ICSM*, 138–147.
[17] H , M. H , R. M. 2001. An overview of program slicing. *Software Focus 2*, 3, 85–92.
[18] H , J., D , M. B., Z , H. 2000. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation 13*, 4, 315–353.
[19] H , M. P. E. W , M. W. 1997. Reduction and slicing of hierarchical state machines. In *ESEC '97/FSE-5: Proceedings of the 6th European Conference held jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Springer, New York, NY, USA, 450–467.
[20] H , T. A., Q , S., R , S. K. 1998. You Assume, We Guarantee: Methodology and Case Studies. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, Volume 1427 of *LNCS*. Springer, 440–451.
[21] H , C.A. R. 1983. An axiomatic basis for computer programming. *Commun. ACM 26*, 1, 53–56.
[22] H , C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall.
[23] H , J. M , P. 2005. Model-checking specifications integrating processes, data and time. In *FM 2005*, Volume 3582 of *LNCS*. Springer, 465–480.
[24] H , S., R , T., B , D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12*, 1, 26–60.
[25] J , D. R , E. J. 1994. A new model of program dependences for reverse engineering. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*. ACM Press, New York, NY, USA, 2–10.
[26] J , G. G , S. 2001. Verification Experiments on the MASCARA Protocol. In *Proceedings of SPIN workshop 2001, Toronto, Canada*, Volume 2057 of *LNCS*. Springer, 123–142.
[27] J , N. D. N , F. 1994. Abstract Interpretation: a Semantics-Based Tool for Program Analysis. In *Handbook of Logic in Computer Science*. Oxford University Press.
[28] K , M. 1995. An overview and comparative classification of program slicing techniques. *J. Syst. Softw. 31*, 3, 197–214.
[29] K , J. 2003. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau.
[30] L , W. J., K , H. N., C , S. D., K , Y. R. 2000. A slicing-based approach to enhance Petri net reachability analysis. *Journal of Research Practices and Information Technology 32*, 2, 131–143.
[31] L , A. D . 2001. Program slicing: Methods and applications. In *First IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, Los Alamitos, California, USA, 142–149.
[32] M , Z. P , A. 1995. *Temporal verification of reactive systems: safety*. Springer, New York, NY, USA.
[33] M , L. I. T , T. 2000. Issues in Slicing PROMELA and Its Applications to Model Checking, Protocol Understanding, and Simulation. *STTT 2*, 4, 343–349.
[34] M , C. 1990. Of wp and CSP. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*. Springer, New York, NY, USA, 319–326.
[35] M¨ -O , M. S , H.. 2001. On Optimal Slicing of Parallel Programs. In *Proceedings of the 33rd annual ACM symposium on Theory of computing*. ACM Press, 647–656.
[36] N , M. G. 2001. *Slicing Concurrent Java Programs: Issues and Solutions*. PhD thesis, Indian Institute of Technology, Bombay.
[37] N , F., N , H. R., H , C. 2005. *Principles of Program Analysis*. Springer.
[38] N , T., P , L. C., W , M. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Volume 2283 of *LNCS*. Springer.

[39] O , T. A , K. 1993. Specification Slicing in Formal Methods of Software Development. In *Proceedings of the 17th Annual International Computer Software and Applications Conference*. IEEE Computer Society Press, 313–319.
[40] O , K. J. O , L. M. 1984. The program dependence graph in a software development environment. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*. ACM Press, 177–184.
[41] O , S., R , J. M., S , N., S , M. K. 1994. A Tutorial on Using PVS for Hardware Verification. In *TPCD '94: Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience*, Volume 901 of *LNCS*. Springer, London, UK, 258–279.
[42] P , D. A. 1998. Ten years of partial order reduction. In *Computer Aided Verification*, Volume 1427 of *LNCS*. Springer, 17–28.
[43] R , A. W. 1998. *The Theory and Practice of Concurrency*. Prentice Hall.
[44] S , G. 2000. *The Object-Z Specification Language*. Kluwer Academic Publisher.
[45] S , G. 1996. Combining Slicing and Constraint Solving for Validation of Measurement Software. In *Static Analysis Symposium*, 332–348.
[46] T , F. 1995. A survey of program slicing techniques. *Journal of programming languages 3*, 121–189.
[47] T , F., C , J.-D., F , J., R , G. 1996. Slicing class hierarchies in C++. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 179–197.
[48] T , H. S , S. 2000. How to Drive a B Machine. In *ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*. Springer, 188–208.
[49] T , H. S , S. 1999. Using a Process Algebra to Control B OPERATIONS. In *IFM'99 1st International Conference on Integrated Formal Methods*. Springer, York, 437–457.
[50] W , M. P. 2002. Program Slicing via FermaT Transformations. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*. IEEE Computer Society, Washington, DC, USA, 357–362.
[51] W , M. 1979. *Program slices: formal psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI.
[52] W , M. 1982. Programmers use slices when debugging. *Commun. ACM 25*, 7, 446–452.
[53] W , J. D , J. 1996. *Using Z — Specification, Refinement, and Proof*. Prentice Hall.
[54] W , F. Y , T. 2004. Slicing Z Specifications. *SIGPLAN Not. 39*, 8, 39–48.
[55] X , B., Q , J., Z , X., W , Z., C , L. 2005. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes 30*, 2, 1–36.

Appendix A.

L 1. (R *wlp*) *For all* $z \in State, tr \in Events^*$ *and all predicates p the following holds:*

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

P . Let $z \in State, tr \in Events^*, p \in ZPred$.

$\Longrightarrow$**(*Correctness*):**

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

I :
$tr = \langle\rangle$ is obvious since $wlp(\langle\,\rangle, p) = p$.
$tr = \langle m.i.o\rangle$ with $m \in Methods$: if $z \overset{tr}{\Rightarrow}_{OZ} z'$, $z \models \mathtt{com}\_m(i, o)$ holds. By definition of wlp we deduce $z' \models p$.

I :
Let the proposition be true for all tr with $\#tr < n$.

I :
Let $tr = \langle m.i.o\rangle \frown tr'$ for $m \in Methods$ and $tr' \in Events^*$ with $\#tr = n$. By definition of wlp we deduce:

$$\begin{aligned} & wlp(tr, p) \\ = \; & wlp(\langle m.i.o\rangle, wlp(tr', p)) \\ = \; & \forall\, State' \bullet (\mathtt{com}\_m(i, o)) \Rightarrow wlp'(tr', p) \end{aligned} \tag{2}$$

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

$\Longrightarrow$(*Completeness*):

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

I :
$tr = \langle\rangle$ is obvious.
$tr = \langle m.i.o\rangle$ with $m \in Methods$: if $z \models \mathtt{com}\_m(i, o)$ holds, by the operational semantics we get $z \overset{\langle m.i.o\rangle}{\Longrightarrow}_{OZ} z'$. This yields that every $m.i.o$-successor of z satisfies p.

I :
Let the proposition be true for all tr with $\#tr < n$.

I :
Again let $tr = \langle m.i.o\rangle \frown tr'$ with $\#tr = n$. We assume

$$\forall\, z' : State' \bullet (z \overset{tr}{\Rightarrow}_{OZ} z') \;\Rightarrow\; (z' \models p) \tag{3}$$

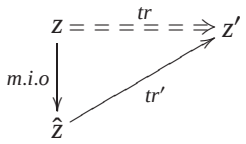
and by definition of wlp have to show that $z \models wlp(\langle m.i.o\rangle, wlp(tr', p))$, which means

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

This means that for every *m.i.o*-successor $\hat{z}$ of z $wlp(tr', p)$ holds. By induction hypothesis applied to tr' it is sufficient to show that the following holds:

$$\forall z' : State' \bullet (\hat{z} \overset{tr'}{\Rightarrow}_{OZ} z') \Rightarrow (z' \models p')$$

We take a look at the following picture:



We know that $z \overset{\langle m.i.o \rangle}{\Longrightarrow}_{OZ} \hat{z}$ and $\hat{z} \overset{tr'}{\Rightarrow}_{OZ} z'$ holds. This yields $z \overset{\langle m.i.o \rangle^\frown tr'}{\Longrightarrow}_{OZ} z'$ and with (3) we deduce $z' \models p$. This completes the proof. □