Slicing Object-Z Specifications for Verification*

Ingo Brückner[1] and Heike Wehrheim[2]

[1] Universität Oldenburg, Department für Informatik, 26111 Oldenburg, Germany
ingo.brueckner@informatik.uni-oldenburg.de
[2] Universität Paderborn, Institut für Informatik, 33098 Paderborn, Germany
wehrheim@uni-paderborn.de

Abstract. Slicing is the activity of reducing a program or a specification with respect to a given condition (the slicing criterion) such that the condition holds on the full program if and only if it holds on the reduced program. Originating from program analysis the entity to be sliced is usually a program and the slicing criterion a value of a variable at a certain program point. In this paper we present an approach to slicing Object-Z specifications with temporal logic formulae as slicing criteria and show the correctness of our approach. The underlying motivation is the goal to substantially reduce the size of the specification and subsequently facilitate verification of temporal logic properties.

1 Introduction

Program slicing has been introduced by Weiser [18, 19] as a technique for reducing programs with respect to some criteria under interest. The title of his first article already suggests what the main idea of slicing and its main application was (and partly still is): "programmers use slices when debugging". Whenever a variable turns out to have a wrong value at a certain program statement programmers are interested in finding out what the part (slice) of the program is which influences this variable value, and for debugging they just want to look at that part. This is exactly what slicing is doing for them. Ever since this first article a huge number of publications on slicing have appeared, introducing slicing techniques in various flavours and for various types of programs (with procedure calls, pointers, concurrency etc.). For a general survey see [15]. Recently, slicing techniques have been transferred to the area of model checking [10, 8] where the slicing criterion is no longer a variable value but a temporal logic formula. In these works slicing should guarantee that the property specified by the formula holds on the reduced program/specification if and only if it holds on the full program/specification. This is similar to the technique of cone-of-influence reduction used in hardware verification [3].

* This work 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). See www.avacs.org for more information.

H. Treharne et al. (Eds.): ZB 2005, LNCS 3455, pp. 415–434, 2005.

In this paper, we suggest a method for slicing Object-Z specifications with respect to formulas of an interval based temporal logic over states and events. To this end a dependence graph of the specification is build which precisely reflects data and control dependencies. Starting from the atomic propositions in the formula this graph is traversed in a backward direction thus determining the part of the specification which potentially influences these atomic propositions. We show that the remaining part of the specification can safely be omitted when checking for the holding of the formula since the formula holds on the full specification iff it holds on its slice. This can substantially reduce the size of the Object-Z specification and thus the state space during verification of temporal logic properties.

A related approach has been presented in [17, 16] where slicing techniques have been used to determine whether changes of a specification might influence already proven properties. In contrast to the work there we will here build a dependence graph with a much finer granularity. Dependencies will be determined on the level of *predicates* not complete schemas. This allows to omit some predicates in a schema while keeping other necessary parts. Moreover, we will use a state- and event-based temporal logic for property specification instead of ordinary LTL. The logic is inspired by the Duration Calculus (DC) [21], however, omitting the time. The reason for choosing a logic talking about events for a state-based formalism lies in our ultimate goal of extending this work to CSP-OZ [7], a combination of Object-Z with the process algebra CSP, and finally to CSP-OZ-DC [9] which additionally adds Duration Calculus formulae (for specifying timing constraints) to CSP-OZ. Properties will in the final setting be expressed in DC (which is one of the reasons for choosing a timeless variant of DC here).

The paper is structured as follows. In the next section we introduce Object-Z (or more precisely, the Object-Z part of CSP-OZ) by means of a small example which we later use for slicing. Furthermore, following Winter and Smith [20] we define a Kripke structure semantics for Object-Z. This is used as the basis for interpreting the temporal logic (SE-IL) formulae. Section 3 will then present the construction of the dependence graph and the slicing algorithm, and illustrate both on the running example. The slicing algorithm will be proven correct with respect to preservation of the SE-IL property under interest in section 4. The last section concludes.

2 Background

This section sets the background for our work on slicing Object-Z specifications. We briefly describe Object-Z [13, 6] by means of an example that we will later use for slicing. Furthermore we introduce the temporal logic and explain how to give a Kripke structure semantics to Object-Z so that the holding of formulae for Object-Z specifications can be defined. Finally, we give a definition of projection, which is the relation used to compare full and reduced specification.

Example. The example is inspired by the Tic-Tac-Toe specification of [5], but has been slightly modified to serve as a good example for slicing. Tic-Tac-Toe is a game involving two players (called black and white) and a board with 9 positions in a 3-by-3 array.

0	1	2
3	4	5
6	7	8

The players take turns to move. A move consists of choosing a free position and adding it to the players owned positions. The goal (in our modified version) is to obtain as many diagonal, vertical or horizontal lines with three positions as possible[3]. The game ends when all positions are occupied.

$$Posn == 0..8$$

The function *inLine* determines whether a set of positions contains a line with three positions, the function *lines* counts the number of three-position lines.

$$inLine : \mathbb{P}\, Posn \rightarrow \mathbb{B}$$

$$\forall\, ps : \mathbb{P}\, Posn \bullet inLine(ps) \Leftrightarrow \exists\, s : \{\{0,1,2\},\{3,4,5\},\{6,7,8\},\{0,3,6,\} \{1,4,7\},\{2,5,8\},\{0,4,8\},\{2,4,6\}\} \bullet s \subseteq ps$$

$$lines : \mathbb{P}\, Posn \rightarrow \mathbb{N}$$

$$\forall\, ps : \mathbb{P}\, Posn \bullet lines(ps) = \#\{x, y, z : Posn \mid \{x, y, z\} \subseteq ps \land inLine(\{x, y, z\}) \bullet \{x, y, z\}\}$$

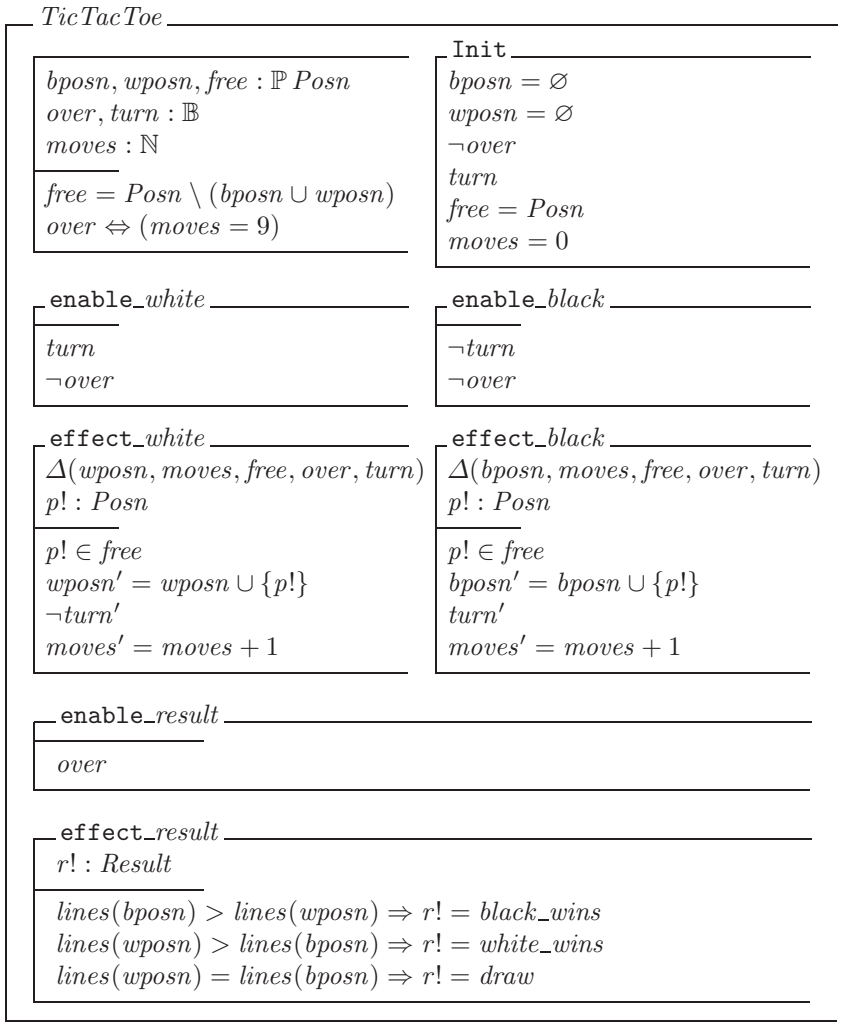
The game has three possible outcomes:

$$Result ::= black\_wins \mid white\_wins \mid draw$$

The following is the specification of the class *TicTacToe*. It is not exactly Object-Z but the Object-Z part of a CSP-OZ [7] specification[4]. The difference can be found in the schemas for methods: in CSP-OZ a method m can be specified by giving an *enable* schema defining a guard to the execution of the method plus an *effect* schema defining the actual execution.

[3] This is the difference to the usual Tic-Tac-Toe where the player with the first line of three positions wins.

[4] Note in particular that no object references are present due to the communications that are used in CSP-OZ. Therefore the aliasing problem does not occur.



This class specification will later be sliced with respect to some temporal logic properties.

Kripke Structure Semantics. The temporal logic will be interpreted on Kripke structures, therefore we will next define a Kripke structure semantics for Object-Z classes. The temporal logic will talk both about states *and* events (viz. methods). In contrast to ordinary Kripke structures transitions are thus labelled with events.

Definition 1. *Let AP be a nonempty set of atomic propositions, E an alphabet of events (or methods names).*

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 $L : S \rightarrow 2^{AP}$.*

An infinite sequence of events and states $s_0 e_1 s_2 e_3 s_4 \ldots$ *is a* path *of the Kripke structure iff* $s_0 \in S_0$ *and* $(s_i, e_{i+1}, s_{i+2}) \in \rightarrow$ *holds for all* $i \geq 0, i$ *even.*

A path is fair *with respect to a set of events* $E' \subseteq E$ *(or* E'-fair*) iff* $inf(\pi) \cap E' \neq \varnothing$ *where* $inf(\pi) = \{e \in E \mid \exists$ *infinitely many* $i \in \mathbb{N} : e_i = e\}$.

By convention we assume that paths are always infinite. This can be achieved by augmenting states s with no outgoing transitions by an extra transition $s \xrightarrow{\tau} s$, where τ is an internal event (e.g. as in the process algebras CSP and CCS). We furthermore will in the following only consider paths that are fair with respect to some E'. This fairness requirement can be seen as an assumption on an environment which infinitely often has to call methods (viz. events) from E'. Since an Object-Z class is not executing methods without a client calling them anyway such fairness requirements are reasonable assumptions.

The Kripke structure semantics for an Object-Z class is obtained by taking all possible valuations of variables as states and state changes via execution of methods as transitions. The set of atomic propositions AP are the predicates over the class' variables, e.g. for class *TicTacToe* predicates $moves = 3$ and $free \neq \varnothing$ are possible atomic propositions. The set of events E are those which can be built from the methods by filling in values for inputs and outputs, e.g. the method *white* gives rise to events *white*.3, *white*.4, etc.. For convenience we will not make an explicit distinction between methods and events here and will not treat inputs and outputs. Thus we say that each class has a set of events E and for every such event there might be an *enable* and an *effect* schema.

Definition 2. *The* Kripke structure semantics *of an Object-Z class* $C = (State, Init, (\texttt{enable\_}e)_{e \in E}, (\texttt{effect\_}e)_{e \in E})$ *is the labelled Kripke structure* $K = (S, S_0, \rightarrow, L)$ *over* AP *and* E *with*

- $S = State$,
- $S_0 = \{s \in S \mid Init(s)\}$ *a set of initial states,*
- *the transition relation* $\rightarrow = \rightarrow' \cup \{(s, \tau, s) \mid \nexists\, s' \ \nexists\, e : s \overset{e}{\rightarrow}' s'\}$ *where*

$$\rightarrow' = \{(s, e, s') \mid \texttt{enable\_}e(s) \wedge \texttt{effect\_}e(s, s')\},$$

- $L(s) = \{p \mid p \in AP \wedge s \Rightarrow p\}$.

In the following we only consider Object-Z classes that satisfy the following two further assumptions: First, we assume the set of initial states to be nonempty ($\exists\, State \bullet Init$) and second, we assume for any *enable* schema to imply the pre-condition of its *effect* schema ($\forall\, e \in E : \texttt{enable\_}e \Rightarrow \text{pre}\, \texttt{effect\_}e$).

Logic. The logic for expressing temporal properties is inspired by the Duration Calculus (DC), and allows us to reason about events and states but (for suiting our purposes) not about time and can therefore be regarded as an untimed projection of DC. There are two reasons for choosing this logic: first of all, our ultimate goal is to apply slicing to integrated specifications which in addition to Object-Z contain parts specifying the dynamic behaviour (in CSP) and timing

constraints (in DC). The logic for expressing properties of this type of specifications will be the full DC. As a second reason, we are interested in a logic which can precisely express orderings between events and state propositions (e.g. like "when event e happens then immediately afterwards variable x has the value 5"). Since we are interested in reducing the specification it should, however, on the other hand not be able to precisely speak about *steps* of the system (e.g. like "the 10th operation of the system is event e"). The paths of the reduced specification will be projections of the paths of the full specification (omitting some events), and thus a preservation of properties under slicing does only make sense for logics which do not talk about particular steps.

The following grammar describes formulae of the state/event interval logic SE-IL (where $ev \in E$ is an event and $p \in AP$ an atomic proposition).

$$
\begin{array}{lll}
\varphi ::= & \lceil p \rceil & \text{– phase (p holds in all states of the given interval)} \\
\mid & ev & \text{– event (ev occurs in the given interval)} \\
\mid & \neg\varphi & \text{– negation} \\
\mid & \varphi \wedge \psi & \text{– conjunction} \\
\mid & \overset{L}{\diamondsuit}\varphi & \text{– eventually operator with liveness (φ holds inside or beyond the given interval)} \\
\mid & \varphi \,;\, \psi & \text{– chop operator (divides the given interval into two parts where φ holds on the first and ψ holds on the second part)}
\end{array}
$$

We use the abbreviation $\overset{L}{\Box}\varphi$ to stand for $\neg\overset{L}{\diamondsuit}\neg\varphi$. For a formula φ we let $E(\varphi)$ denote the set of events occurring in it and $V(\varphi)$ the set of variables of atomic propositions in it.

In order to define when a Kripke structure satisfies an interval logic formula we first define path-satisfaction. Duration Calculus is used to reason about continuous time models, and the validity of formulas is defined via a quantification over all time intervals: a formula holds iff it is true in all intervals (starting at time 0). This definition is now transferred to the discrete setting of paths: a path satisfies a formula iff the formula holds on all intervals $[0, e]$, $e \in \mathbb{N}$. Let $\pi = s_0 e_1 s_2 e_3 s_4 \ldots$ be a path and $\pi[i]$ the i-th component of π: $\pi[i]$ can either be an event or a state.

1. $\pi, [b, e] \models \lceil p \rceil$ iff $\exists\, m, b \leq m \leq e : \pi[m] \in S$
   and $\forall\, m, b \leq m \leq e : \pi[m] \in S \Rightarrow p \in L(\pi[m])$,
2. $\pi, [b, e] \models ev$ iff $b = e$ and $\pi[b] = ev$,
3. $\pi, [b, e] \models \neg\varphi$ iff not $\pi, [b, e] \models \varphi$,
4. $\pi, [b, e] \models \varphi \wedge \psi$ iff $\pi, [b, e] \models \varphi$ and $\pi, [b, e] \models \psi$,
5. $\pi, [b, e] \models \overset{L}{\diamondsuit}\varphi$ iff $\exists\, m_1, m_2 \geq b : \pi, [m_1, m_2] \models \varphi$,
6. $\pi, [b, e] \models \varphi \,;\, \psi$ iff $(\exists\, m, b \leq m \leq e : \pi, [b, m] \models \varphi$ and $\pi, [m, e] \models \psi)$
   $\vee\, (\pi[b] \in S$ and $\pi[b, b-1] \models \varphi$ and $\pi, [b, e] \models \psi)$
   $\vee\, (\pi[e] \in S$ and $\pi[e, e-1] \models \psi$ and $\pi, [b, e] \models \varphi)$

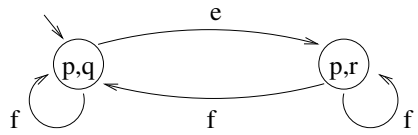
Some explanations for this unusual definition are at place. Item 1: the decision taken here is that during execution of an event we do not know what atomic

propositions hold, thus the formula $\lceil p \rceil$ evaluates to false on an interval with an event only. This reflects the fact that events may invalidate atomic propositions which hold in the state before their execution and make others become true in the state after their execution. In order to be able to say that an event causes a state change we can neither assume that atomic propositions in pre-states still hold while the event takes place nor that those of post-states already hold. Item 2,3 and 4 should be as expected. Item 5: The eventually operator has to reason about positions outside the current interval since we want to achieve real liveness, not just bounded liveness. This operator is taken from the DC with liveness [12]; the standard DC does not allow to reason about unbounded liveness. Item 6: The first part of the disjunction captures the case where the interval is divided into two parts such that φ holds on the first part and ψ on the second. The second and third part of the disjunction mimic the phenomenon that in continuous time one can chop off an empty interval from every interval. The empty interval is denoted by $[b, b-1]$ (or $[e, e-1]$). In an empty interval neither $\lceil p \rceil$ nor ev holds. Note that for instance $ev \,;\, ev \equiv ev$[5] but $\neg\lceil p \rceil \not\equiv \lceil \neg p \rceil$[6].

A Kripke structure then satisfies a formula if all of its paths do (and an Object-Z class satisfies a property when its Kripke structure does).

Definition 3. *Let $K = (S, S_0, \rightarrow, L)$ be a Kripke structure and φ an SE-IL formula. A path π satisfies φ if $\pi, [0, e] \models \varphi$ holds for all $e \in \mathbb{N}$.* K satisfies φ *($K \models \varphi$) iff $\pi \models \varphi$ holds for all paths of K.* K fairly satisfies φ *w.r.t. a set of events $E' \subseteq E$ ($K \models_{E'} \varphi$) iff $\pi \models \varphi$ holds for all E'-fair paths of K.*

As an example consider the following Kripke structure K:



For K we for instance have $K \models \boxed{L} p$ (p always holds), $K \models \neg \diamondsuit_L (e \,;\, \lceil \neg r \rceil)$ (r holds after e, formulated as a counter-example: there is no interval in which $\neg r$ holds immediately after e) but $K \not\models \diamondsuit_L e$ (event e will eventually happen is not true since there are paths with event f only) and $K \not\models \boxed{L} \lceil q \rceil$.

For class *TicTacToe* we are interested in the following two properties:

$$\varphi_1 := \boxed{L} \lceil moves = 9 - \#free \rceil$$
$$\varphi_2 := \neg \diamondsuit_L (black \,;\, (\lceil true \rceil \wedge \neg(\lceil true \rceil \,;\, white \,;\, \lceil true \rceil)) \,;\, black)$$
$$\wedge \neg \diamondsuit_L (white \,;\, (\lceil true \rceil \wedge \neg(\lceil true \rceil \,;\, black \,;\, \lceil true \rceil)) \,;\, white)$$

[5] The formula ev only holds on a zero interval $[b, e]$ with $b = e$. The chop operator can divide this interval into two zero intervals that both satisfy ev.

[6] From the fact that $\lceil p \rceil$ does not hold on an interval one cannot conclude that $\lceil \neg p \rceil$ holds on this interval.

Property φ_1 states an invariant between two variables of the class and φ_2 states that moves are taken in turn. The second property is again formulated as a counter-example: there should not be an interval in which an event *black* is followed by a nonempty interval in which no *white* happens which is then followed by another *black* (and similar for *white*). Nonemptiness of the middle interval is achieved by conjunction with $\lceil true \rceil$.

Projection of Event-Labelled Kripke Structures. The task of slicing is to compute a reduced specification which satisfies a certain property if and only if the full specification satisfies it. For proving this we will show that the reduced specification is a *projection* of the full specification onto some relevant subset of the atomic propositions and events, i.e. they only differ on atomic propositions and events that the formula does not mention.

The projection relation is again first defined on paths and then lifted to Kripke structures. Intuitively, when computing the projection of a given path onto a set of atomic propositions and a set of events one divides the path into blocks such that all states inside a block are "projection-equivalent" (i.e. they coincide on the given set of atomic propositions) and all events inside a block are "irrelevant" events (i.e. events not from the given set of events) except for the last event in the block which is a "relevant" event (i.e. an event from the given set of events). The projection of the original path contains then any path such that for each of the blocks of the original path all states and irrelevant events are mapped onto one single state of the new path while the "relevant" event remains in the new path as illustrated in the following sketch of a projection of a path:

	Block 0	Block 1	Block 2	Block 3
$\pi \;=$	$s_0\ e_0\ s_1\ e_1$	$s_2\ e_2$	$s_3\ e_3\ s_4\ e_4$	$\ldots$
$Pr(\pi)\ \ni$	$r_0\ e_1$	$r_1\ e_2$	$r_2\ e_4$	$\ldots$

Definition 4. *Let $\pi = s_0 e_0 s_1 e_1 s_2 e_2 s_3 \ldots$ be an E'-fair path over a set of atomic propositions AP and a set of events $E \supseteq E'$. The* projection of π onto a set of atomic propositions AP' and a set of events E' ($Pr_{AP',E'}(\pi)$) *contains any E'-fair path $\rho = r_0 f_0 r_1 f_1 r_2 f_2 r_3 \ldots$ such that there is a sequence of indices $0 = i_0 < i_1 < i_2 < \ldots$ (that divides π into blocks) with*

- $\forall\, k \geq 0$: $L(s_{i_k}) \cap AP' = L(s_{i_k+1}) \cap AP' = \cdots = L(s_{i_{k+1}-1}) \cap AP' = L(r_k) \cap AP'$
  (relevant atomic propositions do not change within a block and are the same in the correspondent state of ρ),
- $\forall\, l \in \mathbb{N}, \forall\, k : i_l \leq k < i_{l+1} - 1 : e_k \in E \setminus E'$
  (no relevant events occur inside a block),
- $\forall\, l \geq 1 : e_{i_l - 1} = f_{l-1} \in E'$
  (transitions between blocks are labelled with the same relevant event as the correspondent transition of ρ).

For comparing the Kripke structures we restrict the definition to fair paths since we are only considering satisfaction of formulae on fair paths.

Definition 5. *Let $K_i = (S_i, S_{0,i}, \rightarrow_i, L_i)$, $i \in \{1, 2\}$, be labelled Kripke structures over a set of atomic propositions AP and a set of events E, $AP' \subseteq AP$ a subset of the atomic propositions and $E' \subseteq E$ a subset of the events.*

K_2 is in the projection *of K_1 onto AP' and E' ($K_2 \in Pr_{AP',E'}(K_1)$) iff the following holds:*

1. *For each E'-fair path π in K_1 there exists an E'-fair path π' in K_2 such that $\pi' \in Pr_{AP',E'}(\pi)$,*
2. *and vice versa, for each E'-fair path π' in K_2 there exists an E'-fair path π in K_1 such that $\pi' \in Pr_{AP',E'}(\pi)$.*

Such a projection relation between two Kripke structures guarantees that formulae which only mention propositions from AP' and events from E' hold for either both or none of the Kripke structures.

Theorem 1. *Let φ be an SE-IL formula over AP' and E', and K_1, K_2 labelled Kripke structures over a set of atomic propositions AP and a set of events E with $AP' \subseteq AP$ and $E' \subseteq E$. If $K_2 \in Pr_{AP',E'}(K_1)$ then the following holds:*

$$K_1 \models_{E'} \varphi \quad \textit{iff} \quad K_2 \models_{E'} \varphi \ .$$

Due to space restrictions we omit the proof that can be found in appendix 1 of the full version of the paper [1].

3 Slicing

Slicing means reducing a program or a specification such that the reduced program/specification only contains those parts of the full specification which can influence a certain property called the slicing criterion. At the beginning slicing criteria have usually been of the form "what is the value of variable x at statement n?". The task of the slicing algorithm was to find the (smallest) part of the program/specification sufficient for correctly answering this question.

In the context of model checking slicing criteria have become more complex and are usually temporal logic formulae. Nevertheless, techniques similar to ordinary slicing can be used for slicing with respect to temporal logic formulae since the essence of slicing has remained the same: slicing needs precise information about dependencies between different parts of a program/specification. Such dependencies are represented in a *program (or system) dependence graph*[7]. This section explains the construction of program dependence graphs for Object-Z classes and their slicing with respect to SE-IL formulae.

[7] We stick to the word program although we treat specifications.

Program Dependence Graph. We start with some notational conventions. We assume V to be the set of variables of the class, E to be its methods (or events) and *Pred* to be predicates over V.

For a predicate p over a set of variables $vars(p)$ standing in some schema we define $mod(p)$ to be those variables which occur in primed form and are in the Δ-list of the schema, and $ref(p)$ to be the variables occurring in unprimed form. For input and output variables we use the following convention: output variables of a predicate p are in $mod(p)$ and input variables in $ref(p)$. This effect could alternatively be achieved by embedding inputs and outputs into the state as in [14]. For the initialisation schema *Init* we assume the Δ-list to be V and $ref(p)$ to be $\varnothing$ for all predicates p in *Init* (although variables appear in unprimed form they are actually set in the *Init* schema). For an effect schema `effect_`e and variables u, v we say that u *constrains* the value of v in `effect_`e ($constrains^{\texttt{effect\_}e}(u, v)$) if there is a predicate p in `effect_`e such that both u and v are in $mod(p)$. The relation *constrains* is thus symmetric.

The construction of the program dependence graph (PDG) starts with the construction of the control flow graph (CFG) (depicted in fig. 1). It contains

- one node n_{Init} labelled *Init*,
- one node n_{DO} labelled *DO* (nondeterministic choice),
- for every method/event e two nodes n_{en_e} and n_{eff_e} labelled `enable_`e and `effect_`e.

These nodes also appear in the PDG where they are supernodes, i.e. nodes that contain a number of ordinary nodes. This hierarchical relation corresponds to the relation between predicates (ordinary nodes) that occur inside schemas (supernodes). The control flow between nodes is used to determine dependencies in the PDG.

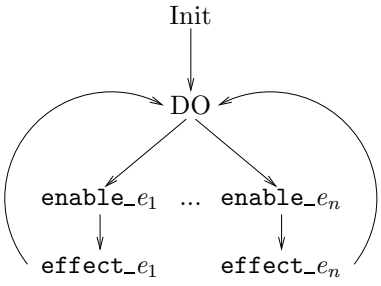


Fig. 1. Control flow graph of a class

The construction of the PDG then proceeds in two steps. The first step is a kind of normalisation (although not as complete as the ordinary one) on the specification; the second step builds the graph.

1. First step: Class normalisation.
   - The state invariant is attached to every effect schema in primed form.
   - For every variable v of type T occurring in the Δ-list of some schema but not in primed form in a predicate we add a predicate $v' \in T$ to the schema (in order to make it explicit that the variable might change).
2. Second step: Graph construction.
   From the CFG we build a hierarchical graph in which the supernodes are those of the CFG and the predicates of the schemas occur as subnodes. Furthermore, we add *control dependencies* between two nodes n and n' if the evaluation of the predicate of n may influence the execution of n', and *data dependencies* if n modifies a variable that n' references.
   More formally, for a class $(State, Init, (\texttt{enable\_}e)_{e\in E}, (\texttt{effect\_}e)_{e\in E})$ we build a hierarchical graph $G = (K, P, l, \rightsquigarrow, \rightarrowtail)$ with
   - $K = \{n_{Init}, n_{DO}\} \cup \{n_{en\_e} \mid e \in E\} \cup \{n_{eff\_e} \mid e \in E\}$ a set of *supernodes*,
   - $P = \{p_x \mid p \text{ is a predicate in a schema named } x\}$, a set of ordinary nodes[8],
   - l a *labelling function* defined as

$$\begin{aligned} l:\ n_{Init} &\mapsto Init \\ n_{DO} &\mapsto DO \\ n_{en\_e} &\mapsto \texttt{enable\_}e \\ n_{eff\_e} &\mapsto \texttt{effect\_}e \\ p_x &\mapsto p \end{aligned}$$

   - $\rightsquigarrow \subseteq P \times P$ the *data dependence* edges defined by $p_x \rightsquigarrow q_y$ iff
     - directed data dependencies exist, i.e.

$$mod(p_x) \cap ref(q_y) \neq \varnothing \text{ and } y \neq Init\ ,$$

     - or symmetric data dependencies exist, i.e.

$$mod(p_x) \cap mod(q_y) \neq \varnothing \text{ and } y = x\ ,$$

   - $\rightarrowtail \subseteq P \times P$ the *control dependence* edges defined by

$$p_x \rightarrowtail q_y \text{ iff } \exists\, e \in E : x = \texttt{enable\_}e \text{ and } y = \texttt{effect\_}e\ .$$

The program dependence graph of the class *TicTacToe* can be found in fig. 2. Control dependencies between two supernodes (of an enable and an effect schema) stand for dependencies between every predicate in the first and in the second node. It can be seen that due to normalisation the effect schemas have two extra predicates which appeared in the original specification as the state invariant.

[8] The hierarchical relation between an ordinary node and its associated supernode is implicitly present in an ordinary node's index that refers to the schema the predicate comes from, i.e. its supernode.

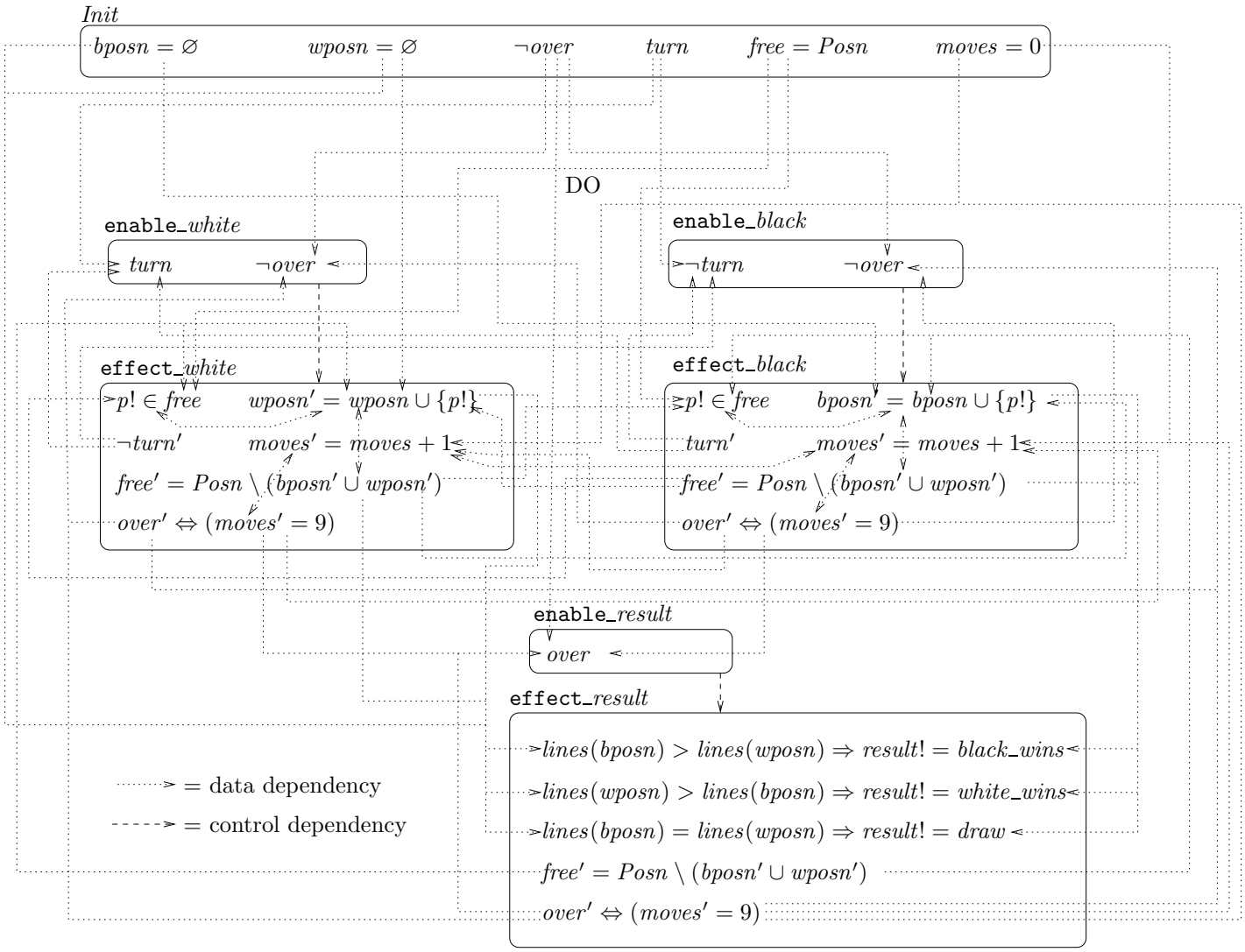


Fig. 2. PDG of class TicTacToe

Backward Slice. The construction of the program dependence graph is independent of the actual SE-IL formula. The formula comes into play when the slicing is carried out. In ordinary slicing the slicing criterion is the value of a variable at a certain program statement. In order to construct the slice of a program w.r.t. this criterion the node representing the statement is determined and then all nodes are included in the slice which are backward reachable (via dependencies) from this particular node. In this way the part of the program which might influence the slicing criterion is obtained.

When slicing w.r.t. SE-IL formulae this is less easy. We first have to find out what the "start nodes" for slicing are, i.e. which nodes represent the slicing criterion. From the formula φ we can derive a set of events E_φ and a set of variables V_φ under interest (those appearing in the formula). From these we can determine the nodes N_φ (predicates) in the PDG which directly manipulate these variables or influence the execution of these events:

$$\begin{aligned} E_\varphi &= E(\varphi) \\ V_\varphi &= V(\varphi) \cup \{v \mid \exists\, e \in E_\varphi, \exists\, p_{en\_e} : v \in vars(p)\} \\ N_\varphi &= \{p_x \mid \exists\, v \in V_\varphi : v \in mod(p)\} \cup \{p_y \mid \exists\, e \in E_\varphi : y = en\_e\} \end{aligned}$$

The nodes in N_φ are those from which the slicing is then started. All nodes in the *backward slice* of N_φ might potentially influence execution of events in E_φ or values of (and thus atomic propositions over) V_φ.

$$bs(N_\varphi) = \{n' \in P \mid \exists\, n \in N_\varphi : n'(\rightsquigarrow \cup \rightarrowtail)^* n\}$$

The backward slice contains the set of nodes which influence the truth value of φ and thus gives us the events, predicates and variables which still have to be in the reduced class specification.

$$\begin{aligned} N' &= bs(N_\varphi) \\ V' &= \bigcup_{p_x \in N'} vars(p) \\ E' &= \{e \mid \exists\, p : p_{en\_e} \in N' \vee p_{eff\_e} \in N'\} \end{aligned}$$

There are, however, some variables in V' whose values cannot influence the holding of the formula since they are never referenced (i.e. never occur in unprimed form in predicates of N'). Thus we define a second set of variables

$$\overline{V} = V_\varphi \cup \{v \in V' \mid \exists\, p_x \in N' : v \in ref(p)\}$$

which are those actually referenced. Variables out of $V' \setminus \overline{V}$ are still needed in the reduced specification since there might be predicates referring to their primed version. As an example, consider an effect schema with predicates $u' = v'$ and $v' = 5$ where $u \in V(\varphi)$. Since the value of u in the post-state is constrained by that of v both predicates and variables are needed in the reduced specification. The value of v in some state is however never used, it cannot influence the value of u. Thus v would be in V' but not in $\overline{V}$. We let $\overline{AP}$ denote the set of atomic propositions over $\overline{V}$.

Reduced Specification. Given the set N', V' and E' it is then straightforward to construct the reduced specification. The class C^{red} contains a state schema with variables from V' only (same type as in C), with schemas only for events in E' (plus *Init*), and in these schemas only the predicates from nodes in N'. We refer to the schemas in this class specification as $State^{red}$, $Init^{red}$, $\texttt{enable\_}e^{red}$, $\texttt{effect\_}e^{red}$ and in order to properly distinguish it from the original specification this will in the following be called C^{full}.

Examples. When slicing the class *TicTacToe* with respect to the formula

$$\varphi_1 := \boxed{L}\lceil moves = 9 - \#free \rceil$$

the result is the following:

$$\begin{aligned}
N' &= N \setminus \{(lines(bposn) > lines(wposn) \Rightarrow result! = black_wins)_{\texttt{effect_}result},\\
&\qquad (lines(wposn) > lines(bposn) \Rightarrow result! = white_wins)_{\texttt{effect_}result},\\
&\qquad (lines(wposn) = lines(bposn) \Rightarrow result! = draw)_{\texttt{effect_}result}\}\\
V' &= V = \overline{V}\\
E' &= E
\end{aligned}$$

Thus the slice w.r.t. φ_1 exhibits only one difference in comparison to the original specification, namely only the predicates are removed that determine the final result that is communicated by schema *result*. This is sensible, of course, since the communicated result does not influence the given property.

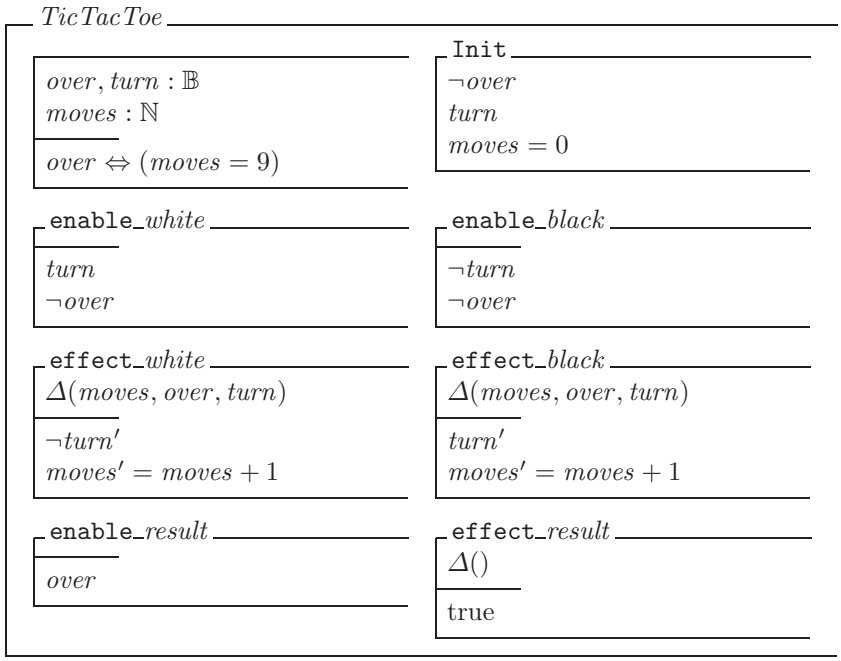
When slicing the class *TicTacToe* with respect to the formula

$$\begin{aligned}
\varphi_2 := {} & \neg \overset{L}{\diamond}(black\ ;\ (\lceil true \rceil \wedge \neg(\lceil true \rceil\ ;\ white\ ;\ \lceil true \rceil))\ ;\ black)\\
& \wedge \neg \overset{L}{\diamond}(white\ ;\ (\lceil true \rceil \wedge \neg(\lceil true \rceil\ ;\ black\ ;\ \lceil true \rceil))\ ;\ white)
\end{aligned}$$

the result is the following:

$$\begin{aligned}
N' = N \setminus \{ & (bposn = \varnothing)_{Init}, (wposn = \varnothing)_{Init}, (free = Posn)_{Init},\\
& (p! \in free)_{\texttt{effect_}white}, (wposn' = wposn \cup \{p!\})_{\texttt{effect_}white},\\
& (free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect_}white},\\
& (p! \in free)_{\texttt{effect_}black}, (bposn' = bposn \cup \{p!\})_{\texttt{effect_}black},\\
& (free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect_}black},\\
& (lines(bposn) > lines(wposn) \Rightarrow r! = black_wins)_{\texttt{effect_}result},\\
& (lines(wposn) > lines(bposn) \Rightarrow r! = white_wins)_{\texttt{effect_}result},\\
& (lines(wposn) = lines(bposn) \Rightarrow r! = draw)_{\texttt{effect_}result},\\
& (free' = Posn \setminus (bposn' \cup wposn')_{\texttt{effect_}result}\}\\
V' = V \setminus \{ & bposn, wposn, free\} = \overline{V}\\
E' = E &
\end{aligned}$$

This leads to the following specification slice:



Thus, additional to the difference that we saw in the previous example, this slice has another difference in comparison to the original specification: All predicates have been removed that determine the sets of free and occupied fields. This difference is sensible since the given property expresses only that there is a strict alternation between the players' moves. In order to analyze the sequence of moves the players can perform, the exact occupation of fields during the course of the game is irrelevant and all related predicates can safely be removed together with the variables that store the associated information about free and occupied fields.

This example shows that slicing can substantially reduce the size of the specification and hence the state space of the associated Kripke structure. Verification of temporal logic properties can thus be facilitated.

4 Correctness

In this section we show correctness of the slicing algorithm, i.e. we show that the Kripke structure of the reduced specification is a projection of that of the full specification. As a consequence the property (and slicing criterion) φ then holds on the full specification if and only if it holds on the reduced specification. In the proofs we use the notation of the last section, i.e. let N', E', V' denote the nodes, events, variables which remain in the specification or PDG after slicing and $\overline{V}, \overline{AP}$ are the variables and atomic propositions, respectively, on which the full and reduced specification should agree.

We start the correctness proof with two lemmas showing the relationships between events, predicates and variables which remain in the specification.

Lemma 1. *Let $e \in E'$ be an event and p a predicate out of the schema* `enable_`e*. Then $p_{en\_e} \in N'$.*

Proof: Take some $e \in E'$. Then by definition of E' there is some predicate q such that either $q_{en\_e} \in N'$ or $q_{eff\_e} \in N'$.

- Assume $q_{en\_e} \in N'$. Then either $q_{en\_e} \in N_\varphi$ or it is not in N_φ but in the backward slice of N_φ. In the first case $e \in E_\varphi$ and hence $p_{en\_e} \in N_\varphi$ and thus in N'. Or q_{en_e} is in the backward slice of N_φ. Since q is coming from an enable schema the outgoing dependencies are only control dependencies. Hence there is some predicate r in `effect_`e such that r is in the backward slice. The control dependency is going from q to r but also from p to r and therefore p is in N' as well.
- Assume $q_{eff\_e} \in N'$. Then p_{en_e} is in the backward slice since there is a control dependency from p_{en_e} to q_{eff_e}.

As a consequence, either all or none of the predicates of an enable schema are in the backward slice.

Corollary 1. $\forall\, e \in E' :$ `enable_`$e^{red} =$ `enable_`e^{full}.

The next lemma shows that events not in E', i.e. omitted in the reduced specification, have no influence on the variables in $\overline{V}$.

Lemma 2. *Let $e \notin E'$ be an event. For all predicates p appearing in* `effect_`e *we then have $mod(p) \cap \overline{V} = \varnothing$.*

Proof: Assume there is some $\overline{v} \in \overline{V}$ with $\overline{v} \in mod(p)$. Then one of the following two cases hold:

1. $\overline{v} \in V_\varphi$
   $\Rightarrow p_{eff\_e} \in N_\varphi$
   $\Rightarrow p_{eff\_e} \in N'$
   $\Rightarrow e \in E'$ (Contradiction!)
2. $\overline{v} \in \{v \in V' \mid \exists\, q_x \in N' : v \in ref(q)\}$
   $\Rightarrow$ data dependency from p_{eff_e} to q_x
   $\Rightarrow p_{eff\_e} \in N'$
   $\Rightarrow e \in E'$ (Contradiction!)

Next, we state the main theorem of the paper which is the correctness of slicing with respect to the interval logic property. This is proven by showing that the Kripke structure of the reduced specification is a projection of that of the full specification.

Theorem 2. *Let C^{full} be a class specification, φ an SE-IL formula and C^{red} the class obtained when slicing C^{full} with respect to φ. Let E' and $\overline{AP}$ be the set of events and atomic propositions, respectively, which the slicing algorithm delivers as those of interest (in particular $E(\varphi) \subseteq E'$ and $V(\varphi) \subseteq \overline{V}$).*

Let furthermore K^{full} and K^{red} be the corresponding Kripke structures. Then the following holds:

$$K^{red} \in Pr_{\overline{AP},E'}(K^{full})$$

Proof:

1. Let $\pi = s_0 e_1 s_2 e_3 \ldots$ be an E'-fair path of K^{full}. We construct a sequence $\rho' = t_0 f_1 t_2 f_3 \ldots$ with

   $t_i : s_i|_{V'}$
   $f_i : e_i$ if $e_i \in E'$, nop else

   Out of ρ' we construct a sequence ρ by eliminating all subsequences of the form $nop\ t_j$.
   We have to show that ρ is an E'-fair path of K^{red}.
   (a) Fairness: ρ contains infinitely many events from E' since π contains them and they are preserved by the construction.
   (b) Path: Since K^{red} contains fewer predicates than K^{full} in the schemas, there are fewer restrictions on values of variables. Hence

   $$Init^{full}(s_0) \Rightarrow Init^{red}(s_0|_{V'})$$
   $$\texttt{enable\_}e^{full}(s_i) \Rightarrow \texttt{enable\_}e^{red}(s_i|_{V'})$$
   $$\texttt{effect\_}e^{full}(s_i, s_{i+2}) \Rightarrow \texttt{effect\_}e^{red}(s_i|_{V'}, s_{i+2}|_{V'})$$

   Furthermore by Lemma 2 we know that $e_i \notin E'$ implies $s_{i-1}|_{\overline{V}} = s_{i+1}|_{\overline{V}}$. If there is now a transition $s_{i-1} \xrightarrow{e_i} s_{i+1}$ in K^{full} with $e_i \in E'$ then a transition $t_j \xrightarrow{e_i} t_{i+1}$ with $j \leq i-1, \forall\, k : j < k < i-1 : e_k \notin E'$ and $e_{j-1} \in E'$ is possible in K^{red} since 1) $t_j|_{\overline{V}} = s_j|_{\overline{V}} = s_{i-1}|_{\overline{V}}$ and $t_{i+1}|_{\overline{V}} = s_{i+1}|_{\overline{V}}$ and 2) $\texttt{enable\_}e_i{}^{red}$ only references variables from $\overline{V}$ and 3) $\texttt{effect\_}e_i{}^{red}$ only references unprimed variables from $\overline{V}$ and moreover makes no or the same restrictions as $\texttt{effect\_}e_i{}^{full}$ on primed variables in V' (and hence the value in s_{i+1} is ok, since it coincides with the value in t_{i+1}).
   Thus ρ is a path of K^{red}.
   By construction ρ is in the projection of π onto $\overline{AP}$ and E'.
2. Let $\rho = t_0 f_1 t_2 f_3 \ldots$ be a (by definition) E'-fair path of K^{red}. We inductively construct a sequence $\pi = s_0 e_1 s_2 e_3 \ldots$ by

   $s_i : s_i \xrightarrow{e_{i+1}}_{full} s_{i+2}$
   $e_i : f_i$

   We have to show the well-definedness of this construction (by induction), i.e. show that a transition $s_i \xrightarrow{e_{i+1}}_{full} s_{i+2}$ is indeed possible in K^{full}. Furthermore we have to show that $s_{i+2}|_{\overline{V}} = t_{i+2}|_{\overline{V}}$ (note that s_{i+2} and t_{i+2} might differ on variables in $V' \setminus \overline{V}$).
   (a) Induction base: s_0
   We know that $Init^{red}(t_0)$ holds. We have to show that there is some s_0 with $Init^{full}(s_0)$ and $s_0|_{\overline{V}} = t_0|_{\overline{V}}$. To this end we show that $Init^{red}$ contains all predicates of $Init^{full}$ which directly or indirectly influence the variables of $\overline{V}$.

i. Directly: let $\overline{u} \in \overline{V}$ and q a predicate in $Init^{full}$ such that $\overline{u} \in mod(q)$. Then $q_{Init} \in N'$ since either 1) $\overline{u} \in V_\varphi$ and then $q_{Init} \in N_\varphi$ and hence in N', or 2) $\overline{u} \in \{v \in V' \mid \exists\, p_x \in N' : v \in ref(p)\}$ and then there is data dependency from q_{Init} to p_x and hence $q_{Init} \in N'$.

ii. Indirectly: let $\overline{u} \in \overline{V}$ and assume there is a chain of variables $u_1, u_2, \ldots, u_n$ such that $u = u_1$, $\overline{u} = u_n$ and $constrains(u_i, u_{i+1}), 1 \leq i \leq n-1$. Due to symmetric dependencies all predicates q causing the *constrains*-relationships are in N'.

(b) Induction step: assume we have constructed the sequence up to some state s_i and $s_i|_{\overline{V}} = t_i|_{\overline{V}}$.
We have to show that e_{i+1} is enabled in s_i and its execution leads to some state s_{i+2} such that $s_{i+2}|_{\overline{V}} = t_{i+2}|_{\overline{V}}$. By Corollary 1 we get $\texttt{enable\_}e_{i+1}{}^{red} = \texttt{enable\_}e_{i+1}{}^{full}$, and thus there is some s'_{i+2} with $s_i \xrightarrow{e_{i+1}}_{full} s'_{i+2}$. Now we choose $s_{i+2} = s'_{i+2} \oplus t_{i+2}|_{V''}$ with

$$V'' = \bigcup_{p_{effect\_e_{i+1}{}^{full}} \in N'} mod(p) \cup \overline{V}.$$

This choice ensures that $s_{i+2}|_{\overline{V}} = t_{i+2}|_{\overline{V}}$ holds since $\overline{V} \subseteq V''$. Furthermore this choice is admissible since each predicate in $\texttt{effect\_}e_{i+1}{}^{full}$ falls into one of the following cases:

- $p_{\texttt{effect\_}e_{i+1}{}^{full}} \in N'$: Then $mod(p) \subseteq V''$ and due to the construction of the slice $ref(p) \subseteq \overline{V}$. Since $s_i|_{\overline{V}} = t_i|_{\overline{V}}$, $s_{i+2}|_{V''} = t_{i+2}|_{V''}$ and $\texttt{effect\_}e_{i+1}{}^{red}(t_i, t_{i+2})$, we know that predicate p holds for the transition from s_i to s_{i+2}.
- $p_{\texttt{effect\_}e_{i+1}{}^{full}} \notin N'$: Then $mod(p) \cap V'' = \varnothing$ holds, since otherwise there would be some variable $v \in mod(p)$ inducing a dependence such that $p_{\texttt{effect\_}e_{i+1}{}^{full}} \in N'$. From $mod(p) \cap V'' = \varnothing$ we know that $s_i|_{mod(p)} = s'_{i+2}$. Together with $\texttt{effect\_}e_{i+1}{}^{full}(s_i, s'_{i+2})$ this leads finally to $\texttt{effect\_}e_{i+1}{}^{full}(s_i, s_{i+2})$.

By construction π is thus an E'-fair path of K^{full} and furthermore ρ is in the projection of π onto $\overline{AP}$ and E'.

As a consequence of theorem 1, the formula φ holds on the reduced specification if and only if it holds on the full specification.

5 Conclusion

This paper is concerned with reducing Object-Z specifications for the verification of temporal logic properties. Given a formula the technique presented in this paper computes a reduced specification on which the formula holds if and only if it holds for the full specification. The technique can substantially facilitate verification of specifications since the preparatory construction of the program dependence graph is only linear in the size of the original specification while its state space is usually much larger (or infinite) and might therefore not be

amenable for an analysis. Slicing can thus be seen as one method for fighting the state explosion problem in verification, along with other techniques like abstraction (for Z for instance by combining the work of [14] and [4]), symmetry reduction, compositional verification (like e.g. [20]) and partial order reductions.

Related Work. Slicing in formal specifications, in particular in Z, has been proposed in [2, 11]. These works carry out slicing with respect to a "standard" slicing criterion, which are the values of variables. Slicing with respect to temporal logic formulae is usually done either in the context of hardware verification [3], therein known as cone-of-influence reduction, or in software model checking, most notably in the Bandera project [8] where it is applied to Java programs.

Future Work. So far, this technique considers a single class only. It could be extended to larger systems either by combining it with compositional verification techniques (e.g. for Object-Z [20]), or by constructing a program dependence graph of the whole system. The latter could be achieved by combining program dependence graphs of the individual objects through a special new dependency arc reflecting the call structure between objects (possibly following approaches for slicing programs with procedures).

The development of tool support for slicing is another important issue. Our small example already revealed the necessity for a program computing program dependence graphs and backward slices and the presented algorithms for these computations clearly suggest such an automation. This is envisaged in the research project AVACS which forms the overall context of this work.

Our main focus for future work is, however, an extension of this technique to an integrated specification formalism combining Object-Z with CSP and Duration Calculus.

Acknowledgement. We would like to thank Jochen Hoenicke for numerous discussions on the definition of a state- and event-based interval logic and for a careful reading of the paper.

References

1. I. Brückner and H. Wehrheim. Slicing Object-Z specifications for verification. Technical Report 3, SFB/TR 14 AVACS, http://www.avacs.org/, 2005.
2. D. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *ACM SIGSOFT international symposium on Software testing and analysis*, pages 138–153. ACM, 1994.
3. E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.
4. J. Derrick and G. Smith. Linear temporal logic and Z refinement. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology (AMAST 2004)*, volume 3116 of *Lecture Notes in Computer Science*, pages 117–131. Springer, 2004.
5. R. Duke and G. Rose. *Formal object-oriented specification using Object-Z*. Macmillan, 2000.

6. R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
7. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
8. J. Hatcliff, M. Dwyer, and H. Zheng. Slicing software for model construction. *Higher-order and Symbolic Computation*, 13(4):315–353, 2000.
9. J. Hoenicke and E.-R. Olderog. Combining Specification Techniques for Processes, Data and Time. In M. Butler, L. Petre, and K. Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 245–266. Springer-Verlag, May 2002.
10. L. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *Software Tools for Technology Transfer*, 2(4):343–349, 2000.
11. T. Oda and K. Araki. Specification slicing in formal methods of software development. In *Proceedings of the Seventeenth Annual International Computer Software & Applications Conference*, pages 313–319. IEEE Computer Society Press, 1993.
12. J. U. Skakkebæk. Liveness and fairness in duration calculus. In B. Jonsson and J. Parrow, editors, *CONCUR'94*, volume 836 of LNCS, pages 283–298. Springer-Verlag, 1994.
13. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publisher, 2000.
14. G. Smith and K. Winter. Proving Temporal Properties of Z specifications Using Abstraction. In *ZB2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 260–279. Springer, 2003.
15. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3(3), 1995.
16. H. Wehrheim. Inheritance of Temporal Logic Properties. In *FMOODS 2003: Formal Methods for Open Object-based Distributed Systems*, number 2884 in LNCS, pages 79–93. Springer, 2003.
17. H. Wehrheim. Preserving Properties under Change. In F.S. de Boer, M. Bonsague, S. Graf, and W.P. de Roever, editors, *Formal Methods for Components and Objects*, volume 3188 of *LNCS*, pages 330–343. Springer, 2004.
18. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
19. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
20. K. Winter and G. Smith. Compositional Verification for Object-Z. In *ZB2003: Formal Specification and Development in Z and B*, number 2651 in LNCS, pages 280–299. Springer, 2003.
21. Zhou Chaochen, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 40/5:269–276, 1991.