Slicing Concurrent Real-Time System Specifications for Verification*

Ingo Brückner

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

Abstract. The high-level specification language CSP-OZ-DC has been shown to be well-suited for modelling and analysing industrially relevant concurrent real-time systems. It allows us to model each of the most important functional aspects such as control flow, data, and real-time requirements in adequate notations, maintaining a common semantic foundation for subsequent verification. Slicing on the other hand has become an established technique to complement the fight against state space explosion during verification which inherently accompanies increasing system complexity. In this paper, we exploit the special structure of CSP-OZ-DC specifications by extending the dependence graph—which usually serves as a basis for slicing—with several new types of dependencies, including timing dependencies derived from the specification's DC part. Based on this we show how to compute a specification slice and prove correctness of our approach.

1 Introduction

When modelling and analysing complex systems, their various behavioural aspects need to be considered such as the admitted sequence of events taking place during operation, the associated modifications induced on the system state, or the real-time constraints that need to be imposed on the system's behaviour in order to achieve a functionally correct system model. In the area of safety-critical systems the application of formal methods with exactly defined semantics is advisable which are open to subsequent analysis and mathematical proof of certain desired properties. However, there is no single formalism which is equally well suited for each of the needed modelling tasks. Therefore, numerous combinations of different such modelling notations have been proposed in order to address each of the different system aspects with a dedicated technique [15, 21, 26, 24].

The notation we consider in this paper is the high-level specification language CSP-OZ-DC [12], a formalism which has already been shown to be appropriate for modelling industrially relevant specifications such as parts of the European Train Control System (ETCS, [8]). CSP-OZ-DC combines three individually well-researched formalisms: *Communicating Sequential Processes* (CSP,

* 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, `www.avacs.org`).

[11]) to specify system behaviour in terms of ordering of events and communication between processes; *Object-Z* (OZ, [20]) to define a system's state space and modifications associated with the occurrence of events; *Duration Calculus* (DC, [9]) to define real-time properties over certain events or states. In CSP-OZ-DC, a common semantic basis is given to these three formalisms by extending the DC semantics such that it also covers the CSP and the OZ part. Furthermore, CSP-OZ-DC provides a compositional translation into phase event automata, a variant of timed automata which is appropriate for subsequent verification by model-checking [8, 19, 2].

One of the main obstacles for automated verification, however, is the problem of state space explosion, i.e., the exponential blow-up in the number of system states to be analysed. Many techniques have been proposed to tackle this problem and the frontier of systems being amenable to model checking has been pushed forward again and again throughout the last years by the—often complementary—application of various state reduction methods such as partial-order reduction [18] or predicate abstraction [6].

Aiming in the same direction is the method of *slicing*. It was originally introduced by Weiser [25] in the context of program analysis in order to determine those parts of a program which are relevant with respect to a specific debugging task. Having become an established method in the area of program analysis [23], slicing has found numerous further areas of application [28] in the past decades, among them the area of *software verification* where it has successfully been applied to various targets such as Java [10] and Promela [17]. As a syntax-based approach that operates at the source code level, slicing can exploit additional knowledge about the system structure. It has hence been shown to be effective in addition to complementary techniques working on the semantic level of the models generated from the source code [7].

Slicing in the context of verification is usually done in two steps. First, a *dependence graph* is constructed representing control and data dependencies present inside the source code. This first preparatory step is independent from the actual verification property and thus only needs to be performed once for a given verification target. Second, a backwards analysis is performed on the dependence graph with the verification property as *slicing criterion*, i.e., a starting point to identify elements that directly or indirectly affect the property to be verified. Computing the specification slice relevant to the verification property allows us to omit the irrelevant parts in the subsequent verification step such that the state space to be examined is already reduced before the verification actually starts. An important requirement in this context is the *correctness* of the slicing approach, i.e., the verification result must remain the same, regardless of whether verification is performed on the original target or on its slice.

In this paper we apply slicing to CSP-OZ-DC specifications as a preparatory step for subsequent verification with respect to *test formulas* [16], which form a subset of DC that is amenable to model-checking. The rich syntactical structure of CSP-OZ-DC specifications and their clear separation in different parts addressing different system aspects makes them an ideal target for the syntax-

oriented technique of slicing. We exploit the special structure of CSP-OZ-DC specifications by introducing several new types of dependencies such as *predicate*, *synchronisation*, and *timing* dependencies into the dependence graph. In comparison to conventional dependence graphs these dependencies yield additional information about the specification allowing us to construct a more precise dependence graph and thus a more precise slicing outcome. Building upon previous work [4, 3], we show correctness of our approach not only with respect to test formulas, but, more generally, with respect to any logic which is invariant under stuttering, i.e., which cannot distinguish between interpretations that are equivalent up to some stuttering steps (defined by sets of irrelevant variables and events obtained from slicing).

The paper is structured as follows. The next section introduces CSP-OZ-DC by giving a small example and roughly introducing the semantics of such specifications. In section 3 we present the dependence graph construction and the subsequent slicing algorithm, both illustrated by the running example. A sketch of the correctness proof of the slicing algorithm is given in section 4. The last section concludes and briefly discusses related work.

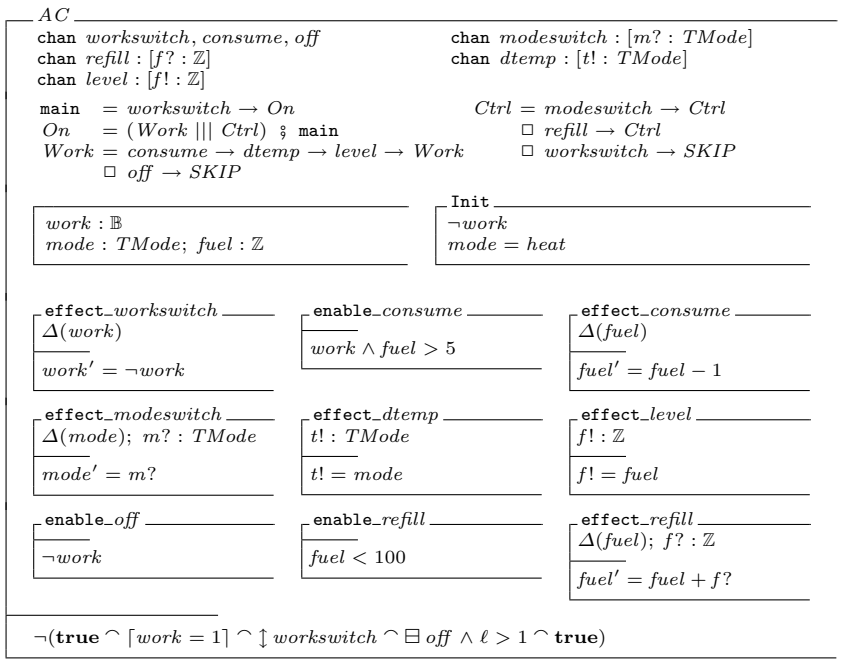
2 CSP-OZ-DC

For illustrating our approach we use a CSP-OZ-DC specification of an air conditioner system. It can operate in two modes, either heating or cooling. Initially the air conditioner is off. When it is switched on (*workswitch*), it starts to run. While running, the air conditioner either heats or cools its environment and simultaneously allows the user to switch the mode (*modeswitch*), refill fuel (*refill*) or switch it off again. Cooling or heating is modelled by a consumption of one unit of fuel (*consume*) and an emission of hot or cold air (*dtemp*). For the specification we first define the mode of operating ($TMode ::= heat \mid cool$).

The first part of the class defines its interface towards the environment by means of several communication channels (`chan`). The next part specifies its dynamic behaviour, i.e., the allowed ordering of method execution. It is defined via a set of CSP process equations, beginning with the initially active `main` process. The operators appearing here are prefixing $\rightarrow$, sequential composition $\fatsemi$, interleaving $|||$ (parallel composition with no synchronisation) and external choice $\Box$. The third part of a CSP-OZ-DC class describes the attributes and methods of the class.

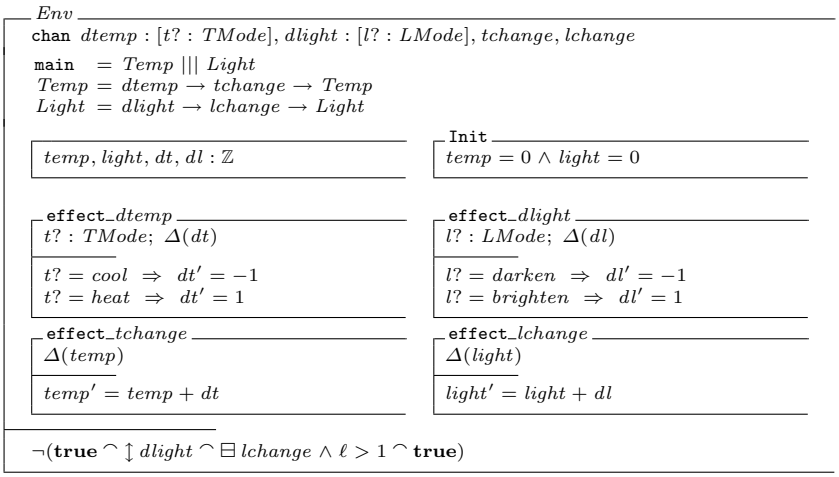
For every method we might have an `enable` schema fixing a guard for method execution (enabling schemas equivalent to *true* are left out) and an `effect` schema describing the effect of a method upon execution. For instance, the `enable` schema of method *consume* tells us that the air conditioner has to be on and a minimal amount of fuel is necessary for *consume* to take place. Upon execution one unit of fuel is consumed according to its `effect` schema. The method *level* on the other hand is always enabled, it just displays the current level of fuel.

The concluding *Duration Calculus* (DC) part of the class defines real-time properties of the system within a number of DC *counterexample formulas*, i.e., a subset of DC which is amenable for later verification. The only formula in the DC part of AC specifies that whenever the air conditioner is turned on for some time ($\lceil work = 1 \rceil$) and an event *workswitch* occurs, an event *off* must follow within at most one time unit (negatively defined by $\ldots \frown \boxminus off \wedge \ell > 1 \frown \ldots$ as part of the counterexample). The expression $\lceil work = 1 \rceil$ denotes a non-empty time interval throughout which predicate $work = 1$ holds. $\updownarrow workswitch$ refers to a point interval at which event *workswitch* occurs, while $\boxminus off$ refers to a non-empty time interval without any occurrence of event *off*. The chop operator $\frown$ connects all three intervals and surrounds them with initial and concluding **true** intervals of arbitrary length. These surrounding intervals enable the observation of the counterexample to happen at any point in time.



The air conditioner's environment is specified within a second class. Apart from modelling the temperature, this class also models the lighting situation (via type $LMode ::= brighten \mid darken$), possibly determined by some further components beyond the scope of our small example.

Intuitively, it is already quite obvious that in this specification the additional aspect of lighting is completely independent from the temperature. In section 3 we will see how to automatically obtain this observation as part of the slicing result.



Finally, parallel composition of the air conditioner and the environment with synchronisation on the set of common events defines our complete example system:

$$System = AC \underset{\{dtemp\}}{\parallel} Env$$

The compositional semantics of such specifications [12] integrates the trace semantics for CSP [11], the history semantics for Object-Z [20], and the set of interpretations for Duration Calculus formulas [9].

Definition 1. *An* interpretation *is a function* $\mathcal{I}$*:* $Time \to Model$ *mapping the time domain* $Time == \mathbb{R}^+$ *to the set of Z models* $Model == NAME \twoheadrightarrow \mathbb{W}$ *with* $NAME$ *being the set of all valid identifiers and* $\mathbb{W}$ *being the world, i.e., the set of all possible semantic values.*

An interpretation of a CSP-OZ-DC class defines a set of *observables*, i.e., time-dependent functions yielding valuations for

- all variables that are used in the CSP-OZ-DC class,
- boolean *channel variables* for all channels of the CSP-OZ-DC class changing its value at each point in time when the associated event occurs,
- *parameter variables* for all channels equipped with parameters containing the parameter values at the point in time when the associated event occurs.

The following definition yields an abstract view of interpretations where time is not taken into account.

Definition 2. *Let* $\mathcal{I}(t)$ *be an interpretation, changing its valuation at points in time* $0 = t_0 < t_1 < t_2 < \ldots$ *from* M_{i-1} *to* M_i *due to events* e_i *occurring at* t_i*,* $i \geq 1$*. Then* $\mathsf{Untime}(\mathcal{I}) = \langle M_0, e_1, M_1, e_2, M_2, \ldots \rangle$ *is the corresponding sequence of alternating states and events.*

An interpretation is fair *with respect to a set of events* $E' \subseteq Events$ *(or* E'-fair*) iff* $inf(\mathsf{Untime}(\mathcal{I})) \cap E' \neq \varnothing$ *where* $inf(\mathsf{Untime}(\mathcal{I})) = \{e \in Events \mid \exists \text{ infinitely many } i \in \mathbb{N}\colon e_i = e\}$.

The semantics of a CSP-OZ-DC class is provided by the set of interpretations that satisfy the given class, i.e., by interpretations $\mathcal{I}$ that satisfy all three parts comprising the class.

CSP part: $\mathcal{I} \models C_{CSP}$ iff $\mathsf{Untime}(\mathcal{I})$ corresponds to a run of the labelled transition system that is defined by the operational semantics of the CSP part [11].

Object-Z part: $\mathcal{I} \models C_{OZ}$ iff $\mathsf{Untime}(\mathcal{I})$ is in the history semantics of the Object-Z part [20], i.e., its first valuation satisfies the *Init* schema of the Object-Z part, all its valuations satisfy the *State* schema of the Object-Z part, and all its events together with their pre- and post-states satisfy the `enable` and `effect` schemas of the associated method.

DC part: $\mathcal{I} \models C_{DC}$ iff $\mathcal{I}$ satisfies each of the DC formulas according to the semantics of DC [9].

To argue about the events taking place at a given point in time, we use the following function.

Definition 3. *Let* $\mathcal{I}\colon \mathit{Time} \to \mathit{Model}$ *be an interpretation and* t *a point in time.* $\mathsf{TakesPlace}(\mathcal{I}, t)$ *is the set of events that take place in* $\mathcal{I}$ *at time* t*:*

$$\mathsf{TakesPlace}(\mathcal{I}, t) = \{e \in \mathit{Events} \mid \exists\, \varepsilon > 0\colon \forall\, t_l \in [t-\varepsilon, t), t_r \in [t, t+\varepsilon]\colon \mathcal{I}(t_l)(e) \neq \mathcal{I}(t_r)(e)\}$$

The next definition allows us to refer to the CSP process term that remains in a given interpretation at a given point in time.

Definition 4. *Let* `main` *be the CSP part of a CSP-OZ-DC specification* C *and* $\mathcal{I}$ *an interpretation satisfying* C *with* $0 = t_0 < t_1 < t_2 < \ldots$ *the points in time where* $\mathcal{I}$ *changes and* $e_i \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ *for* $i > 0$*. Then the* residual CSP process term *associated with a point in time, denoted by* $\mathsf{CSP}_C(\mathcal{I}, t_i)$*, is defined as* $\mathsf{CSP}_C(\mathcal{I}, t_i) = P_i$ *with* $\texttt{main} \equiv P_0 \xrightarrow{e_1} P_1 \xrightarrow{e_2} \ldots \xrightarrow{e_i} P_i$ *being a valid transition according to the operational semantics of the CSP part of* C*.*

For describing properties of CSP-OZ-DC classes we can now use DC test formulas [16] which can be evaluated on the set of interpretations defined by the CSP-OZ-DC class. In this paper, we will not introduce this subset of DC, but instead only assume that our logic is invariant under projection, i.e., that it cannot distinguish interpretations where one is a projection of the other onto some set of observables. A precise definition of projection is given in section 4.
One property of interest for our air conditioner specification could for instance be whether there are always at least 5 fuel units left when the air conditioner is on (which in fact is not true): $\varphi \equiv \neg\Diamond(\lceil \mathit{work} \wedge \mathit{fuel} < 5 \rceil)$.
The main purpose of the technique proposed in this paper is to determine whether it is possible to check the property on a reduced specification C' such that the following holds[1]: $C \models \varphi$ iff $C' \models \varphi$.
As we will see it is possible to omit elements of the CSP part, the Object-Z part, and of the DC part of the example system for checking our property.

[1] $C \models \varphi$ stands for "the formula φ holds on all interpretations satisfying C".

3 Slicing

In general, the aim of program slicing is to determine those parts of a given program that are relevant with respect to a given slicing criterion such that only these relevant parts need to be considered when analysing the program with respect to the slicing criterion. This relevance analysis is usually based on the preceding construction of a program dependence graph (PDG) that comprises all dependence relations between elements of the program code. In preparation for the construction of the PDG for CSP-OZ-DC specifications we first construct the specification's control flow graph (CFG) which represents the execution order of the specification's schemas according to the CSP part.

3.1 Control flow graph

Starting with the *start*.`main` node, representing the beginning of control flow according to the CSP `main` process definition, its nodes ($n \in N_{CFG}$) and edges ($\longrightarrow_{CFG} \subseteq N_{CFG} \times N_{CFG}$) 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 `enable_`e and `effect_`e),
- to operators in the CSP part (like nodes *interleave* and *uninterleave* for operator $|||$, nodes *extchoice* and *unextchoice* for operator $\Box$, or nodes par_S and $unpar_S$ for operator $\underset{S}{\|}$), or
- to the structuring of the CSP process definitions (like *start*.P and *term*.P for entry and exit points of CSP process P, or *call*.P and *ret*.P for call and return points of references to process P).

For multiple occurrences of Object-Z methods inside the CSP process definitions unique CFG nodes are introduced, e.g. by a naming convention of the associated `enable` and `effect` nodes where the methods' names are extended by an ordinal referring to their syntactical occurrence inside the CSP process definitions.

Parallel composition of several classes. When computing the PDG for the parallel composition of several classes, we start by constructing the CFG's for each individual class. These are then combined into one single global CFG for the entire parallel composition in the following steps:

1. The CFG nodes *start*.`main` and *term*.`main` of class C are renamed into *start*.C and *term*.C such that these nodes remain unique in the final CFG.
2. For each pair of classes (C_1, C_2) that should run in parallel composition, parallel synchronisation nodes par_S and $unpar_S$ are created and linked to the respective *start* and *term* nodes of each CFG. The synchronisation set S contains all events over which both classes need to synchronise.
3. Finally, new *start*.`main` and *term*.`main` nodes are created and connected to each of the newly created parallel synchronisation nodes.

Instead of constructing one PDG for each individual class as explained in the following section, the construction of the PDG for the parallel composition of all involved classes is then based on this previously constructed global CFG. Apart from this the construction for parallel composition of classes proceeds as usual.

3.2 Program dependence graph

The conventional program dependence graph (PDG) usually represents data and control dependencies that are present inside a program. In our case we derive several additional types of dependence from the rich syntactical structure of CSP-OZ-DC specifications, among them *predicate dependence* representing connections between schemas and associated predicates, *synchronisation dependence* representing mutual communication relations between processes, and *timing dependence* representing timing relations derived from DC formulas.

In addition to the nodes of the CFG, the PDG contains nodes for each predicate inside a specification schema: $N_{pred} = \{p_x \mid p \text{ predicate of schema node } x\}$. Again, predicate nodes are replicated for each occurrence of their associated event inside the CSP part, e.g., for each CFG schema node. Thus the set of nodes of the PDG is $N_{PDG} = N_{CFG} \cup N_{pred}$. Another important difference between both graphs is the set of edges they have. An edge connects two PDG nodes, if predicate, control, data or synchronisation dependencies exist between these nodes according to the definitions given below.

Before continuing with the construction of the PDG we first introduce some abbreviations. When reasoning about paths inside the CFG, we let $path_{CFG}(n, n')$ denote the set of sequences of CFG nodes that are visited when walking along CFG edges from node n to node n'. When we refer to schemas or predicates associated with a PDG node n, we let

- $out(n)$ denote all output variables (those decorated with a !),
- $in(n)$ denote all input variables (those decorated with a ?),
- $mod(n)$ denote $out(n)$ plus all variables being modified (those appearing in the Δ-list of the schema or in primed form in a predicate),
- $ref(n)$ denote $in(n)$ plus all referenced (unprimed) variables
- $vars(n) = mod(n) \cup ref(n)$ denote all variables.

Next, we proceed with definitions of the various kinds of dependence types that establish the program dependence graph for CSP-OZ-DC specifications. An example of each type of dependence can be found in the dependence graph of our example air conditioner system, depicted in figure 1.

Predicate dependence. Each predicate that occurs in a CSP-OZ specification is located inside some schema. The idea of *predicate dependence* edges

$$\xrightarrow{pred} \subseteq (N_{CFG} \times N_{pred} \cup N_{pred} \times N_{CFG})$$

is to represent this relation between schemas and their associated predicates. An example can be seen in figure 1 between node eff_dtemp_10 and node t?=cool => dt'=-1.

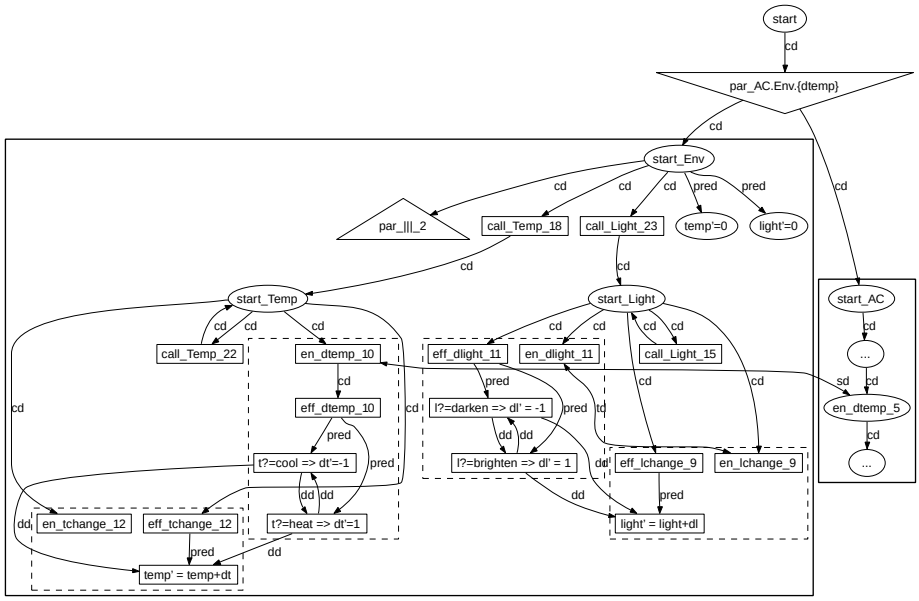


Fig. 1. Program dependence graph for the example system. Nodes inside bold bounding rectangles belong to the same class, nodes inside dashed bounding rectangles to the same event. Note, that most of the AC part is hidden, indicated by "..." nodes.

For predicates of `enable` schemas, these edges lead from the `enable` node to its predicates and vice versa, while for predicates of `effect` schemas there are only edges in the direction from the `effect` schema to its predicates. The different treatment of `enable` and `effect` schema predicates provides a way to represent the tighter connection between `enable` schemas and its predicates: `enable` predicates do not only depend on the event they are associated with but also serve as the event's guard, i.e., a mutual dependence exists, while this is not the case for events of an `effect` schema.

Predicate nodes belonging to the `Init` schema are attached to the associated *start*.`main` node in the way like predicate nodes belonging to an `effect` schema are attached to the associated `effect` node. This reflects the initial restriction of the state space according to the `Init` schema.

Finally, another type of predicate dependence exists for predicate nodes $n \equiv p_x$ implying modifications of variables mentioned in the DC part. Their so far unidirectional connection via the predicate dependence edge coming from their associated effect schema node $n' \equiv$ `effect_`x needs to be complemented by another predicate dependence edge in the opposite direction. This treats such predicate nodes in a similar way as predicate nodes of `enable` schemas, since they play—in conjunction with the DC formula—a similar role: They can be regarded as a guard for the associated event, since it can only take place if the predicate complies with the restrictions given in DC.

Control dependence. The further construction of the PDG starts with the introduction of *control dependence* edges: $\xrightarrow{cd} \subseteq N_{CFG} \times N_{CFG}$
The idea behind these edges is to represent the fact that an edge's source node controls whether the target node will be executed. In particular, a node cannot be control dependent on itself. We distinguish the following types of control dependence edges:

- Control dependence due to *nontrivial precondition* exists between an `enable` node and its `effect` node iff the `enable` schema is non-empty (i.e., not equivalent to true).
- Control dependence due to *external (resp. internal) choice* or *parallel composition with synchronisation* exists between an *extch* (resp. *intch*) or par_S node and its immediate CFG successors.
- Control dependence due to *synchronisation* exists between an `enable` node and its associated `effect` node iff both nodes are located inside a branch attached to a parallel composition node and their associated event belongs to the synchronisation alphabet of this parallel composition node. Note, that even an event with an empty `enable` schema can be source of a control dependence edge, since synchronisation determines whether control flow continues.
- Control dependence due to *timing* exists between an `enable` node and its associated `effect` node iff there exists a DC formula that mentions the given event or variables that are modified by it. Again, even events with an empty `enable` schema can be source of a control dependence edge, since the DC part may restrict whether control flow continues.

An example of control dependence due to synchronisation can be seen in figure 1 between nodes en_dtemp_10 and eff_dtemp_10.

Additionally, some further control dependence edges are introduced in order to achieve a well-formed graph:

- *Call* edges exist between a *call* node and its associated *start* node.
- *Termination* edges exist between a *term* node and its associated *ret* node.
- *Start* edges exist between a *start* node and its immediate CFG successor.
- *Return* edges exist between a *ret* node and its immediate CFG successor.

Finally, all previously defined (direct) control dependence edges are extended to CFG successor nodes as long as they do not bypass existing control dependence edges. The idea of this definition is to integrate indirectly dependent nodes (that would otherwise be isolated) into the PDG.

- *Indirect control dependence* edges exist between two nodes n and n' iff

$$\exists\, \pi \in path_{CFG}(n, n') \colon\ \forall\, m, m' \in \operatorname{ran} \pi : m \xrightarrow{cd} m' \Rightarrow m = n$$

An example of indirect control dependence can be seen in figure 1 between nodes start_Light and en_lchange_9.

Data dependence. The idea of *data dependence* edges $\xrightarrow{dd} \subseteq N_{pred} \times N_{pred}$ is to represent the influence that one predicate might have on a different predicate by modifying some variable that the second predicate references. Therefore, the source node always represents a predicate located inside an `effect` schema, while the target node may also represent a predicate located inside an `enable` schema. We distinguish the following types of data dependence edges:

- *Direct data dependence* exists between two predicate nodes p_x and q_y (appearing in schemas x and y) iff there is a CFG path between both associated schema nodes without any further modification of the relevant variable, i.e., iff

$$\exists\, v \in (\mathit{mod}(p_x) \cap \mathit{ref}(q_y))\,, \exists\, \pi \in \mathit{path}_{CFG}(x, y) \colon \\ \forall\, m \in \operatorname{ran} \pi \colon v \in \mathit{mod}(m) \Rightarrow (m = x \vee m = y)$$

- *Interference data dependence* exists between two nodes p_x and q_y iff the nodes of both associated schemas x and y are located in different CFG branches attached to the same interleaving or parallel composition operator, i.e., iff $\mathit{mod}(p_x) \cap \mathit{ref}(q_y) \neq \varnothing$ and $\exists\, m \colon (m \equiv \mathit{interleave} \vee m \equiv \mathit{par}_S)$ with

$$\exists\, \pi_x \in \mathit{path}_{CFG}(m, x) \wedge \exists\, \pi_y \in \mathit{path}_{CFG}(m, y) \colon \operatorname{ran}\ \pi_x \cap \operatorname{ran}\ \pi_y = \{m\}$$

- *Symmetric data dependence* exists between two nodes p_x and q_y iff they are associated with the same schema and share modified variables, i.e., iff

$$\mathit{mod}(p_x) \cap \mathit{mod}(q_y) \neq \varnothing \wedge x = y$$

- *Synchronisation data dependence* exists between two predicate nodes p_x and q_y iff both are located inside `effect` schemas whose respective `enable` schemas are connected by a synchronisation dependence edge as defined below and one predicate has an output that the other predicate expects as input, i.e., iff $x = \texttt{effect\_}e \wedge y = \texttt{effect\_}e \wedge \mathit{out}(p_x) \cap \mathit{in}(q_y) \neq \varnothing$

An example of direct data dependence can be seen in figure 1 between nodes t?=cool => dt'=1 and temp'=temp+dt, where the modification of variable *dt* at the source node may directly reach the reference of this variable at the target node.

Synchronisation dependence. The idea of *synchronisation dependence* edges $\xleftrightarrow{sd} \subseteq N_{CFG} \times N_{CFG}$ is to represent the influence that two `enable` schema nodes of the same event have on each other by being located inside two different branches of a parallel composition operator that has the schemas' associated event in its synchronisation alphabet. *Synchronisation dependence* exists between two nodes n and n' with $n \equiv n' \equiv \texttt{enable\_}e$ iff $\exists\, m \equiv \mathit{par}_S$ with $e \in S$:

$$\exists\, \pi \in \mathit{path}_{CFG}(m, n) \wedge \exists\, \pi' \in \mathit{path}_{CFG}(m, n') \colon \operatorname{ran}\ \pi \cap \operatorname{ran}\ \pi' = \{m\}$$

An example of synchronisation dependence can be seen in figure 1 between node en_dtemp_5 and node en_dtemp_10 which both belong to the synchronisation alphabet of AC and *Env*. If one of both events is relevant, this also applies to the other one, since both need to agree in order to occur.

Timing dependence. The representation of dependencies arising from the DC part needs some additional preparation. The idea of *timing dependence* edges $\overset{td}{\longleftrightarrow} \subseteq N_{PDG} \times N_{PDG}$ is to represent the mutual influence between neighbouring elements of a DC counterexample formula.

According to [12], each formula DC occurring in the DC part of CSP-OZ-DC specifications can be represented as a sequence of *PhaseSpec* data structures:

$$DC \mathrel{\widehat{=}} PhaseSpec_0^{DC};\ PhaseSpec_1^{DC};\ \ldots;\ PhaseSpec_n^{DC}$$

with $PhaseSpec_i^{DC}$ comprising all information specified in the associated phases of DC, i.e., invariants, time bounds, and forbidden or desired events. Dependencies are defined between nodes associated with the same or neighbouring *PhaseSpec*s. For each formula DC we then define a timing node sequence (TNS_{DC}) consisting of PDG nodes with relevance to the given formula, i.e.,

- predicate nodes implying modifications of variables mentioned in DC,
- `enable` nodes of events mentioned in DC, and
- the *start*.`main` node of the given class if the initial phase of DC has a time bound different from 0.

The nodes in each TNS_{DC} are ordered according to the syntactical occurrence of their associated specification elements inside DC with the exception of the *start*.`main` node. This node does never occur directly inside a DC formula but rather serves as a reference point for the length of the first phase which is the reason why this node—if present—will appear as the first element of the timing node sequence.

$$\begin{aligned} &n \in \operatorname{ran} TNS_{DC} \ \Leftrightarrow \\ &\quad n \equiv start.\texttt{main} \wedge timebound(PhaseSpec_0^{DC}) > 0 \\ &\quad \vee \exists\, PhaseSpec_i^{DC} : mod(n) \cap vars(PhaseSpec_i^{DC}) \neq \varnothing \\ &\qquad\qquad\qquad \vee\ n \equiv \texttt{enable\_}e \wedge e \in events(PhaseSpec_i^{DC}) \end{aligned}$$

Based on these timing node sequences, bidirectional *timing dependence* exists between two nodes n and n' iff there is a TNS_{DC} with two neighbouring timing nodes n and n'.
An example of timing dependence is the edge between nodes en_dlight_11 and en_lchange_9 in figure 1. This timing dependence is derived from the DC formula

$$DC \equiv \neg(\mathbf{true} \frown \updownarrow dlight \frown \boxminus lchange \wedge \ell > 1 \frown \mathbf{true})$$

which appears in the DC part of the environment specification and which relates both involved events *dlight* and *lchange*. This DC counterexample formula refers to initial and concluding intervals of arbitrary length (**true**), represented by $PhaseSpec_0^{DC}$ and $PhaseSpec_2^{DC}$. In between, it refers to a point interval ($\updownarrow dlight$) followed by a non-empty interval ($\boxminus lchange \wedge \ell > 1$) which are both represented by a single data structure, $PhaseSpec_1^{DC}$ with information about initial events (*dlight*), forbidden events (*lchange*), and interval length ($\ell > 1$).

3.3 Backward slice.

For our purpose, slicing is used to determine that part of the specification that is directly or indirectly relevant for the property φ to be verified and which therefore needs to remain in the specification. Computation of this slice starts from the set of events E_φ and the set of variables V_φ that appear directly in φ. Based on this *slicing criterion* $(E_\varphi,\ V_\varphi)$ we determine the set of PDG nodes with direct influence on φ, i.e., the set of predicate nodes modifying variables from V_φ and the set of `enable` nodes belonging to events from E_φ:

$$N_\varphi = \{p_x \in N_{pred} \mid mod(p_x) \cap V_\varphi \neq \varnothing\} \ \cup\ \{\texttt{enable\_}e \in N_{CFG} \mid e \in E_\varphi\}$$

Starting from this initial set of nodes we compute the backward slice by a reachability analysis of the PDG. The resulting set of backwards reachable nodes contains all nodes that lead via an arbitrary number of predicate, control, data, synchronisation or timing dependence edges to one of the nodes that already are in N_φ. In addition to all nodes from N_φ, the backward slice contains therefore also all PDG nodes with indirect influence on the given property, i.e., it is the set of all relevant nodes for the specification slice:

$$N' = \{n' \in N_{PDG} \mid \exists\, n \in N_\varphi \colon n'\ (\xrightarrow{pred} \cup \xrightarrow{cd} \cup \xrightarrow{dd} \cup \xrightarrow{sd} \cup \xrightarrow{td})^*\ n\}$$

Thus relevant events are those associated with nodes from N' that represent `enable` or `effect` schemas

$$E' = \{e \mid \exists\, n \in N' \colon n \equiv \texttt{enable\_}e \vee n \equiv \texttt{effect\_}e\}$$

and relevant variables are those associated with nodes from N' that represent predicates:

$$V' = \bigcup_{p_x \in N'} vars(p_x).$$

3.4 Reduced specification.

Slicing concludes with the computation of the reduced specification, i.e., a version of the full specification without all details which are not relevant for the property that served as the slicing criterion. Verification with respect to this property can afterwards be performed on this reduced specification while the verification result will be the same.

To compute the CSP part of the reduced specification we need a notion of projection of CSP process definitions onto the set of relevant events:

Definition 5. *Let P be the right side of a process definition from the CSP part of a specification and E be the set of events that appear in the specification. The projection of P with respect to a set of events $E' \subseteq E$ is inductively defined:*

1. $SKIP|_{E'} := SKIP$ *and* $STOP|_{E'} := STOP$

2. $(e \rightarrow P)|_{E'} := \begin{cases} P|_{E'} & \text{if } e \notin E' \\ e \rightarrow P|_{E'} & \text{else} \end{cases}$
3. $(P \circ Q)|_{E'} := P|_{E'} \circ Q|_{E'}$ *with* $\circ \in \{;\ , |||, \sqcap, \Box\}$
4. $(P \underset{S}{\parallel} Q)|_{E'} := P|_{E'} \underset{S \cap E'}{\parallel} Q|_{E'}$

The projection of the complete CSP part is defined by applying the above definition to the right side of each process definition.

Given the set N', V' and E' it is then straightforward to construct the reduced specification. For each class C its slice C' contains

- only channels from E'
- the projection of the original specification's CSP part onto E',
- a state schema with variables from V' only (same type as in C),
- schemas only for events in E' (plus *Init*),
- inside these schemas only predicates associated with nodes in N', and
- a DC part with only counterexample formulas that mention variables from V' and events from E'. (Note that due to the construction of timing dependence edges, for any given counterexample formula either all or none of its variables and events belong to the slice.)

When computing the slice of the complete system, i.e., the air conditioner specification in parallel composition with the environment with respect to the verification property $\varphi \equiv \Diamond(\lceil work \wedge fuel < 5 \rceil)$, we obtain the following results:

AC: Method *level* has been removed, which is sensible, since communicating the current amount of fuel (*level*) does not influence φ. Note that methods *modeswitch*, *dtemp* as well as variable *mode* have not been removed. The reason is that method *dtemp* belongs to the synchronisation alphabet, resulting in a control dependence edge due to synchronisation. However, when computing a slice of the air conditioner alone (without parallel composition with the environment), methods *dtemp* and *modeswitch* together with variable *mode* can be removed, since the amount of available fuel does not depend on the mode of operating.

Env: Methods *tchange*, *dlight* and *lchange* have been removed as well as variables *light*, *temp*, and *dl* and DC formula $\neg\Diamond(\updownarrow dlight \frown \boxminus lchange \wedge \ell > 1)$. This result is also sensible, since the actual effect imposed on the environment's temperature (*tchange* and *temp*) does not influence the verification property and the modelling of the environment's lighting behaviour (*dlight*, *lchange*, *light* and *dl*) is not related to the verification property at all.

To summarise, the specification's state space has not only been reduced with respect to its control flow space (events *level*, *tchange*, *dlight*, and *lchange*) but also with respect to its data state space (variables *light*, *temp*, and *dl*) and its timing requirements (the DC part of *Env*).

Note, that in both cases neither the original nor the sliced specification satisfy the given property, so the verification result will be negative in both cases. Nevertheless, this is exactly what we wanted to achieve: A specification slice must satisfy a slicing criterion if and only if the original specification does so.

4 Correctness

In this section we show correctness of the slicing algorithm, i.e., we show that the property (and slicing criterion) φ holds on the full specification if and only if it holds on the reduced specification. For proving this we will show that an interpretation of the reduced specification is a *projection* of an interpretation of the full specification onto some relevant subset of the variables and events, i.e., they only differ on variables and events that the formula does not mention.

Intuitively, when computing the projection of a given interpretation onto a set of relevant variables and a set of events, one divides the interpretation into blocks formed by time intervals beginning at one relevant event and ending at the next relevant event. The corresponding block in the projection refers to the same time interval, but does not contain any of the irrelevant events that may appear inside the block of the original interpretation. Furthermore, throughout both blocks the interpretation and its projection coincide in the valuation of all relevant variables.

Definition 6. *Let O' be a set of observables, $E' = O' \cap$ Events the set of events within O' and $\mathcal{I}, \mathcal{I}'$ be two E'-fair interpretations with $0 = t_0 < t_1 < t_2 < \ldots$ and $0 = t'_0 < t'_1 < t'_2 < \ldots$ the points in time where $\mathcal{I}$ and $\mathcal{I}'$ change, respectively. $\mathcal{I}'$ is in the* projection of $\mathcal{I}$ with respect to O'*, denoted by* $\mathsf{Projection}_{O'}(\mathcal{I})$*, iff*

1. $\forall\, t\colon \mathcal{I}|_{O'}(t) = \mathcal{I}'|_{O'}(t)$
2. $\forall\, i \geq 0\colon\ \exists\, j\colon (t_i = t'_j \wedge \mathsf{TakesPlace}(\mathcal{I}, t_i) = \mathsf{TakesPlace}(\mathcal{I}', t'_j))$
   $\vee\ (t'_j < t_i < t'_{j+1} \wedge \mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E' = \varnothing)$

Given a logic which is invariant under projections, such a projection relationship between any two interpretations then guarantees that formulas which only mention observables from O' hold for either both or none of the interpretations. Note that projection is a particular form of stuttering.

Correctness proof. Now we start the actual correctness proof with several lemmas showing the relationships between CSP processes and events and variables which remain in the specification. Due to space restrictions we only present the main ideas of the proofs.The complete proofs can be found in [1].

Our first lemma states that the projection of each residual CSP process associated with a projection interval without relevant events as defined in definition 6 can mimic the behaviour of the residual CSP process associated with the last state of the projection block, i.e., the relevant event at the end of the block is enabled at any point inside the block when computing the CSP projection.

Lemma 1 (Transitions of CSP process projections). *Let $P_j, \ldots, P_{j+k+1}$ be CSP processes, E' a set of relevant events, $e_{j+1}, \ldots, e_{j+k-2}$ irrelevant events ($\notin E'$), and e_{j+k} a relevant event ($\in E'$), such that*

$$P_j \xrightarrow{e_{j+1}} P_{j+2} \xrightarrow{e_{j+3}} \ldots \xrightarrow{e_{j+k-2}} P_{j+k-1} \xrightarrow{e_{j+k}} P_{j+k+1}$$

is a valid transition sequence. Then the following holds[2]*:*

$$P \xrightarrow{e_{j+k}} P_{j+k+1}|_{E'} \text{ with } P \in \{P_j|_{E'}, \ldots, P_{j+k-1}|_{E'}\}$$

Proof sketch: The proof builds up on another lemma considering the case of a single CSP transition: Either this transition is labelled with a relevant event $e \in E'$ or with an irrelevant event $e \notin E'$. In the former case it is easy to see that the associated projection also can perform this event e, while in the latter case some further considerations lead to the conclusion that the associated projection will finally perform the same relevant event as the original process. Both cases are shown by induction over the structure of the respective CSP processes. For the proof of the present lemma we then only need to combine both cases in an induction over the length of the projection block and come to the desired result.

Next, we bridge the gap between transition sequences that we can observe for CSP processes and paths that are present in the associated control flow graph.

Lemma 2 (CSP transition sequences and CFG paths). *Let C be a class specification, CFG its control flow graph, $\mathcal{I}$ an interpretation satisfying C with $0 = t_0 < t_1 < t_2 < \ldots$ the points in time where $\mathcal{I}$ changes, t_i with $i > 0$ one of these points with $e \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ and $f \in \mathsf{TakesPlace}(\mathcal{I}, t_{i+1})$. Then the two corresponding nodes* `enable_`e *and* `enable_`f *of CFG are related in either one of the following ways:*

1. *There exists a path in CFG which leads from* `enable_`e *to* `enable_`f*:*

   $$path_{CFG}(\texttt{enable\_}e, \texttt{enable\_}f) \neq \varnothing$$

2. *There exists a CFG node $interleave^i$ or par^i_S with $S \cap \{e, f\} = \varnothing$ which has* `enable_`e *and* `enable_`f *as successors in different branches:*

   $$\begin{aligned} &\exists\, n \in CFG\colon n \equiv interleave^i \vee (n \equiv par^i_S \wedge S \cap \{e,f\} = \varnothing) : \\ &\quad \exists\, \pi_e \in path_{CFG}(n, \texttt{enable\_}e) \wedge \exists\, \pi_f \in path_{CFG}(n, \texttt{enable\_}f) : \\ &\qquad \pi_e \cap \pi_f = \{n\} \end{aligned}$$

Proof sketch: The proof consists of two layers of induction over the structure of the residual CSP process terms $\mathsf{CSP}_C(\mathcal{I}, t_i)$ and $\mathsf{CSP}_C(\mathcal{I}, t_{i+1})$ such that each possible combination of CSP constructs is shown to be covered by one of the two cases mentioned in the lemma.

The following lemma states that the set of irrelevant events appearing inside a projection block does not have any influence on the relevant variables associated with the states inside the block.

Lemma 3 (No influence of irrelevant events on relevant variables). *Let C be a class specification, $\mathcal{I}$ an interpretation satisfying C with $0 = t_0 < t_1 < t_2 < \ldots$ the points in time where $\mathcal{I}$ changes, associated with $e_i \in \mathsf{TakesPlace}(\mathcal{I}, t_i)$ for $i > 0$. Let furthermore E' be the set of relevant events computed by the slicing*

[2] Note, that $P_j|_{E'} = \ldots = P_{j+k-1}|_{E'}$ does not necessarily hold.

algorithm with respect to some formula φ (with an associated set of variables V_φ), $e_{j+1}, \ldots, e_{j+k-1} \notin E'$, and $e_j, e_{j+k} \in E'$. Then the following holds:

$$\mathcal{I}(t_j)|_{\overline{V}} = \ldots = \mathcal{I}(t_{j+k-1})|_{\overline{V}} \quad \textit{with } \overline{V} = V_\varphi \cup \bigcup_{e \in \{e_i \in E' \mid i \geq j\}} ref(e)$$

Proof sketch: We show this by contradiction: Supposed, the equality does not hold. This implies the existence of a data dependence between an event inside the block and the relevant event. In consequence, this leads to the event inside the block being a member of the set of relevant events.

Our last lemma states that DC formulas which the slicing algorithm identified to be irrelevant with respect to a property to be verified do not impose restrictions on any relevant event.

Lemma 4 (No influence of irrelevant DC formulas on relevant events). *Let C be a class specification, E' the set of relevant events obtained from slicing C with respect to some slicing criterion φ, and DC a counterexample formula from the DC part of C which is irrelevant with respect to φ. Let*

$$E_{DC} = events(DC) \cup \{e \in Events \mid mod(e) \cap vars(DC) \neq \varnothing\}$$

be the set of events that DC refers to either directly or indirectly by referring to some variable that is modified by the respective event. Then the following holds:

1. *There exists no CFG path connecting events from E_{DC} with events from E'.*
2. *Timings of events from E_{DC} are not affected by timings of events from E'.*

Proof sketch: We show both claims by contradiction: Supposed, a path as in (1) exists, then this leads to the existence of a control dependence and thus to DC being relevant. Supposed, an irrelevant event as in (2) is forced to occur before a certain relevant event, then this leads to a connection between both nodes either via control flow edges, via a common DC formula, or via two DC formulas with a common reference point and thus in all cases to e and hence DC being relevant.

Now we come to our main theorem that states the existence of a projection relationship between any two interpretations associated with the original and to the sliced specification.

Theorem 1. *Let C be a class specification and C' the class obtained when slicing C with respect to a formula φ, associated with sets of events E_φ and variables V_φ. Let E' and V' be the set of events and variables, respectively, which the slicing algorithm delivers as those of interest (in particular $E_\varphi \subseteq E'$ and $V_\varphi \subseteq V'$). Then for any E'-fair interpretation $\mathcal{I}$ satisfying C there is a corresponding E'-fair interpretation $\mathcal{I}'$ satisfying C' such that*

$$\mathcal{I}' \in \mathsf{Projection}_{V' \cup E'}(\mathcal{I}).$$

Proof sketch: We need to consider two directions: (1) We have to show that for any interpretation of C we can construct a corresponding interpretation of C' and (2) vice versa. For both directions we define a set of variables $\overline{V}_i$ that contains all variables mentioned in the slicing criterion and for each $e_i \in \mathsf{TakesPlace}(\mathcal{I}, t_i) \cap E'$ all variables referenced by e_i or subsequent relevant events:

$$\overline{V}_i = V_\varphi \cup \bigcup_{e \in \{e_j \in E' | j \geq i\}} ref(e)$$

1. Let $\mathcal{I}$ be an interpretation satisfying C. We inductively construct an interpretation $\mathcal{I}'$ which coincides with $\mathcal{I}$ on relevant relevant events from E' and relevant variables from $\overline{V}_i$, i.e., intervals of $\mathcal{I}$ containing only irrelevant events correspond to intervals of $\mathcal{I}'$ containing no events but the same valuations of relevant variables.
   We have to show that $\mathcal{I}'$ satisfies C'. To this end we use induction over the length of $\mathcal{I}'$ where we apply lemma 3 and lemma 1 when showing that we can remove some intermediate sequences from the original interpretation such that all schemas, process definitions and timing constraints from the reduced specification are satisfied.
2. Let $\mathcal{I}'$ be an interpretation satisfying C' with $0 = t_0 < t_1 < t_2 < \ldots$ the points in time where $\mathcal{I}'$ changes and $\mathsf{TakesPlace}(\mathcal{I}', t_i) \cap E' \neq \varnothing$.
   We inductively construct an interpretation $\mathcal{I}$ with

   $$0 = t_0 < t_0^1 < t_0^2 < \ldots < t_0^{n_0} < t_1 < t_1^1 < t_1^2 < \ldots < t_1^{n_1} < t_2 < t_2^1 < \ldots$$

   the points in time where $\mathcal{I}$ changes, such that the same relevant events appear in $\mathcal{I}$ and $\mathcal{I}'$ at points in time t_i for $i > 0$, and additional (irrelevant) events appear in $\mathcal{I}$ at points in time $t_i^{j_i}$ for $i \geq 0$ and $1 \leq j_i \leq n_i$.
   In the induction we apply lemma 3 to show that we can safely insert the necessary additional steps in $\mathcal{I}$ such that the associated schemas of the full specification are satisfied. Furthermore, we apply lemma 2 to show that these additional steps are possible according to the process definitions from the full specification. Finally, we use lemma 4 to show that the additional DC formulas are satisfied by choosing appropriate points in time for the additional events in $\mathcal{I}$ such that $\mathcal{I}$ is indeed an interpretation of C.

5 Conclusion

We presented a slicing approach with the intention to use it as a preprocessing step in the verification of high-level specifications of concurrent real-time systems with respect to real-time properties. The overall aim of introducing slicing into the verification workflow is to complement other strategies to fight the problem of state space explosion. Our slicing algorithm is custom-tailored to the integrated specification language CSP-OZ-DC in order to exploit its particular features in the construction of an adequate dependence graph. Once this graph is constructed, it allows us to compute slices of the original specification with

respect to a wide set of verification properties as we demonstrated for a small example specification. Subsequent verification runs can be performed on the slice instead of the full specification without changing the verification result.

Currently, we are integrating the proposed slicing technique as a plugin into the modelling environment Syspect [22] which was already used to automatically generate the dependence graph in figure 1. This tool gives (1) a precise CSP-OZ-DC semantics to a subset of UML notations such as state charts, class diagrams, and component diagrams, and (2) has a plugin-based connection to the verification tool chain for CSP-OZ-DC proposed in [13] and evaluated in [8], currently based on the abstraction-refinement model checker ARMC [19] and the deductive model checker SLAB [2].

Related work. *Program slicing* as originally defined by Weiser in the field of program analysis and debugging [25] has been enhanced with respect to many different aspects, having found numerous additional fields of application at the same (for overview papers see [23, 28]) and a similarly wide spectrum of targets, including Z-based specifications [5, 27] 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 [7] have shown that slicing can indeed effectively complement other strategies such as predicate abstraction [6] and partial order reduction [18] 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 be applied beforehand as a relatively cheap syntax-based method to reduce the input to model generation. Thus, the benefit of slicing can be seen in two directions: First, it enables an optimisation 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.

Existing approaches to static slicing of formal specifications, however, do not consider verification, i.e., slicing is not carried out with respect to verification properties. Work on slicing used for reducing programs before verification has for instance been done for Java [7] and Promela [17]. Furthermore, we are not aware of any existing approaches that consider slicing of high-level specifications of *real-time* systems, while on the semantic level of timed automata slicing has been applied in [14].

References

1. I. Brückner. Slicing CSP-OZ-DC Specifications for Verification. Technical report, Univ. Oldenburg, `http://csd.informatik.uni-oldenburg.de/~ingo/ifm07.pdf`, 2007.
2. I. Brückner, K. Dräger, B. Finkbeiner, and H. Wehrheim. Slicing Abstractions. In *FSEN'07*, LNCS. Springer, 2007. to appear.

3. I. Brückner and H. Wehrheim. Slicing an Integrated Formal Method for Verification. In *ICFEM'05*, volume 3785 of *LNCS*, pages 360–374. Springer, 2005.
4. I. Brückner and H. Wehrheim. Slicing Object-Z Specifications for Verification. In *ZB'05*, volume 3455 of *LNCS*, pages 414–433. Springer, 2005.
5. D. Chang and D. Richardson. Static and Dynamic Specification Slicing. In *SIGSOFT ISSTA*, pages 138–153. ACM, 1994.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *CAV'00*, pages 154–169, 2000.
7. M. B. Dwyer, J. Hatcliff, M. Hoosier, V. Ranganath, R. Wallentine, and T. Wallentine. Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs. In *TACAS'06*, 2006.
8. J. Faber and R. Meyer. Model Checking Data-Dependent Real-Time Properties of the European Train Control System. In *FMCAD'06*, pages 76–77. IEEE, 2006.
9. M. R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 9:283–330, 1997.
10. J. Hatcliff, M. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Higher-order and Symbolic Computation*, 13(4):315–353, 2000.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. J. Hoenicke. *Combination of Processes, Data, and Time*. PhD thesis, Univ. of Oldenburg, 2006.
13. J. Hoenicke and P. Maier. Model-checking specifications integrating processes, data and time. In *FM'05*, volume 3582 of *LNCS*, pages 465–480. Springer, 2005.
14. A. Janowska and P. Janowski. Slicing Timed Systems. *Fundamenta Informaticae*, 60(1–4):187–210, 2004.
15. B. Mahony and J. S. Dong. Timed communicating Object-Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, 2000.
16. R. Meyer, J. Faber, and A. Rybalchenko. Model Checking Duration Calculus: A Practical Approach. In *ICTAC 2006*, volume 4281 of *LNCS*, pages 332–346, 2006.
17. L. Millett and T. Teitelbaum. Issues in Slicing Promela and its Applications to Model Checking. *STTT*, 2(4):343–349, 2000.
18. D. A. Peled. Ten years of partial order reduction. In *CAV'98*, number 1427 in LNCS, pages 17–28. Springer, 1998.
19. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL'07*, 2006.
20. G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.
21. G. Smith and I.J. Hayes. An introduction to Real-Time Object-Z. *Formal Aspects of Computing*, 13(2):128–141, 2002.
22. Syspect. Endbericht der Projektgruppe Syspect. Technical report, Univ. of Oldenburg, `http://syspect.informatik.uni-oldenburg.de/`, 2006.
23. F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.
24. H. Treharne and S. A. Schneider. Communicating B Machines. In *ZB'02*, volume 2272 of *LNCS*, pages 416–435. Springer, 2002.
25. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, 1982.
26. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of Circus. In *ZB'02*, volume 2272 of *LNCS*, pages 184–203. Springer, 2002.
27. F. Wu and T. Yi. Slicing Z Specifications. *SIGPLAN*, 39(8):39–48, 2004.
28. B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *SIGSOFT SEN*, 30(2):1–36, 2005.