Deductive Verification for Improving Slicing of Integrated Formal Specifications*

Ingo Brückner
ingo.brueckner@informatik.uni-oldenburg.de

Björn Metzler
bmetzler@uni-paderborn.de

September 19, 2005

1 Introduction

The combination of the two well known formal specification techniques CSP [Hoa85] for *behavioural* aspects of systems and Object-Z (OZ) [Smi00] for *data aspects* of systems into the specification language CSP-OZ [Fis97] has already been subject of intense research. An important challenge, especially when trying to automatically or semi-automatically analyse such specifications, is their inherent complexity, which quickly goes beyond the scope of current analysis techniques such as model checking.

In order to tackle this problem on a different level, we have applied the technique of *program slicing* [Tip95, HDZ00] to CSP-OZ specifications [Brü04, BW05]. The basic idea is to reduce a given specification by eliminating some of its components in such a way that its semantics remains unchanged with respect to a specific verification property (the *slicing criterion*). The subsequent model checking can then work on an easier target: the specification slice. The reduction is based on a preceding analysis of the specification that is very similar to well known program analysis techniques [NNH05]. More specifically, it includes the construction of a *specification dependence graph* (SDG), which comprises – similar to a program dependence graph [HRB90] – all relevant kinds of dependencies between specification elements such as control or data dependencies.

The present work aims at a further improvement of slicing by combining it with deductive verification techniques, applied to certain parts of CSP-OZ specifications. Based on these additional arguments, some control dependencies can be eliminated from the SDG. This can in consequence lead to smaller and more precise specification slices. Our main contribution here is to show, to which specification parts such deductive verification techniques can be applied and how the necessary arguments can be found and used to improve slicing.

*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).

2 Specifying with CSP-OZ

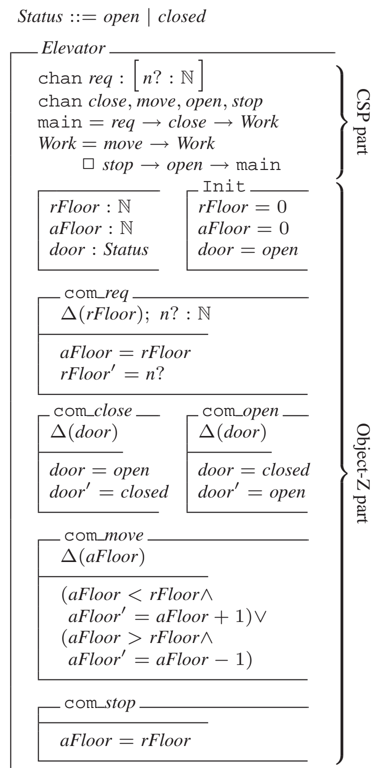

Figure 1: Elevator specification

We introduce CSP-OZ by specifying an example elevator system as depicted in Fig. 1. The specification's CSP part defines the elevator's behaviour. This is done by introducing a set of channels that define the communication with its environment. How this communication looks like is defined in two CSP processes: First (`main`), the passenger requests to be delivered to a certain floor (*req*), upon which the door *close*s and the elevator starts to *Work*. As long as the designated floor has not been reached, the elevator continues to *move*. Once the elevator has reached its destination, it *stop*s, the doors *open* and the system restarts.

The system's state space, its initial configuration and operations on it are then defined in the specification's Object-Z part by so called schemas. Each schema consists of two parts: The upper part may contain a Δ-list of variables that are modified in the lower part and a list of input and output parameters that are used in the lower part for incoming or outgoing communications. The lower part can contain preconditions, which determine whether the associated operation is enabled or not, and predicates, which describe the effect that the associated operation has on the state space.

3 Slicing

To compute the slice of a specification in the context of verification means to compute a reduced specification that exhibits – from the point of view of the verification property – the same behaviour as the original specification. Our approach to achieve this goal for CSP-OZ specifications [BW05] consists of two main steps:
(1) The specification is analysed with respect to the *control* and *data* dependencies it contains, resulting in a *specification dependence graph* (SDG) as depicted in Fig. 2 for the *Elevator* specification[1]. This first step is completely independent of the property to be verified.

(2) Based on the property to verify (e.g. a temporal logic formula), an initial set of SDG nodes is determined that influence the verification property directly. This set of nodes represents the *slicing criterion* that serves as the starting point for a backwards reachability analysis of the SDG. All nodes that cannot be reached backwards from the slicing criterion are without any influence on the verification property. Thus the specification elements that are represented by these nodes can safely be removed from the original specification without changing its semantics with respect to the verification property. The advantage is, that now the sliced specification can be analysed instead of the full specification, and although control and data state space have been reduced, the model checking result holds for both specifications: The verification property is satisfied by the reduced specification if and only if it is satisfied by the full specification [BW05].

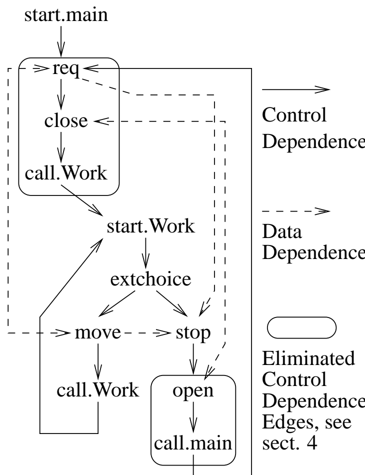

Figure 2: Elevator SDG

Applied to the example specification, our slicing algorithm unfortunately does not achieve any reduction, regardless of which verification property serves as the slicing criterion. The reason for this can easily be seen in the SDG: Regardless of which node we take as the starting point, we can always reach all other nodes backwards via control dependencies. Thus, what we need here is a way to eliminate some of these control dependencies, while the resulting slice must of course remain correct. In the following sections we will sketch how to achieve exactly this by applying deductive verification techniques and how it will be useful for slicing the specification.

[1]Control dependence edges represent the fact that a source node determines if control flow reaches a target node. Data dependence edges represent the fact that a variable modification in a source node might reach a target node.

4 Control Loop Invariants

In the previous section we identified the SDG control dependence edges to be responsible for the failing of our slicing algorithm. These edges represent the fact that their source nodes determine whether control flow reaches the target node. From nodes, which represent events, a control dependence edge can only originate if the associated schema has a non-trivial precondition, i.e. a precondition not equivalent to *true* such that the event can block when the precondition is not satisfied. The main idea of our approach is now to find an argument, which ensures that this precondition is always satisfied at the event's execution, such that it never blocks.

We do this similar to the approach that [TS00] applied to prove deadlock freedom of CSP $\|$ B specifications, where the existence of *control loop invariants*[2] (CLI's) leads to the result of deadlock freedom of the complete specification.

Instead of finding one set of CLI's for the entire specification, we try to find a set of CLI's for each event e and each of its occurences in the specification's CSP part (supposed that e does not already have a trivially satisfied precondition). Assume the CSP part to have processes $Proc := \{P_0, \ldots, P_n\}$ with $P_0 \equiv \mathtt{main}$ and let $Q \in Proc$ be a process in which the respective event occurs. For each process $P_j \in Proc$ we define CLI's that need to satisfy three conditions:
(1) The CLI for the process `main` holds initially:

$$\mathtt{Init} \Rightarrow \mathrm{CLI}_{\mathtt{main}}$$

(2) The CLI for Q implies that the event's precondition is satisfied whenever the event is about to be executed:

$$\mathrm{CLI}_Q \Rightarrow wlp(tr, \mathrm{pre}\, e),$$

where tr is any trace that leads from the start of Q to e. *wlp* (*weakest liberal precondition*) denotes a predicate, which describes that after execution of such a trace the precondition of e is satisfied.
(3) We assume that any right hand side of a process $P_i \in Proc$ leads to a recursive call of another process $P_j \in Proc$ (in case not, there would be no loops containing this process). If the CLI for P_i initially holds, execution of the right hand side of this process implies that the CLI for P_j is satisfied:

$$(CLI_{P_i} \wedge P_i \stackrel{tr}{\Longrightarrow} P_j) \Rightarrow wlp(tr, CLI_{P_j})$$

[2]CLI's are predicates on the state space of the specification. They must hold at each call of their associated process to establish continuity. The term *control* indicates that the CSP part controls the order of execution of the Object-Z schemas.

We have proven a theorem stating that the existence of CLI's satisfying these conditions is sufficient to ensure that the respective event is always enabled.

For event *req* in the *Elevator* specification we compute the following conditions for the CLI's:

(1) $\mathtt{Init} \Rightarrow \mathrm{CLI}_{\mathtt{main}}$

(2) $\mathrm{CLI}_{\mathtt{main}} \Rightarrow (aFloor = rFloor)$

(3) $(\mathrm{CLI}_{Work} \wedge Work \overset{\langle stop, open\rangle}{\Longrightarrow} \mathtt{main}) \Rightarrow \mathrm{CLI}_{\mathtt{main}}$

A basic idea to find a solution for this problem is to use the right hand side of (2) for $\mathrm{CLI}_{\mathtt{main}}$. In addition, this predicate has to be weak enough to be implied by `Init` and also by an appropriate CLI_{Work} added to the precondition and effect of the trace $tr := \langle stop, open\rangle$. In this case, *tr* can only be executed if $aFloor = rFloor$ holds. Since this property is not changed by the effect of *tr*, implication (3) holds without adding a nontrivial CLI for *Work*. One possible solution for all conditions is the following set of CLI's:

$\mathrm{CLI}_{\mathtt{main}} \equiv (aFloor = rFloor)$

$\mathrm{CLI}_{Work} \equiv \mathrm{true}$

These CLI's thus ensure that *req* is enabled at any execution allowed by the specification's CSP part. From the fact that *req* never blocks we infer that there exists no real control dependence originating from *req*, since in spite of its non-trivial precondition *req* does not determine whether control flow reaches the subsequent node *close* or not. Therefore we can remove the associated control dependence edge $req \rightarrow close$ from the original SDG and replace it by an edge $start.main \rightarrow close$ from *req*'s predecessor *start.main* to *close* such that the refined SDG remains well-defined as indicated in Fig. 2.

For event *close* and *open* we can similarly find CLI's satisfying the conditions, so their associated control dependence edges can also be removed from the SDG.

For event *move* we need to find CLI's that satisfy amongst others these conditions:

(1) $\mathrm{CLI}_{Work} \Rightarrow (aFloor \neq rFloor)$

(2) $(\mathrm{CLI}_{Work} \wedge Work \overset{\langle move\rangle}{\Longrightarrow} Work) \Rightarrow \mathrm{CLI}_{Work}$

In this case no solution can be found, the associated edge can not be removed. The same holds for *stop*.

5 Improved Slicing

The previously described approach leads to a significantly reduced SDG. To illustrate the effect of this reduction on the slicing outcome, we compute the slice for property $\Box\Diamond(aFloor = rFloor)$, stating that the designated floor is eventually equal to the current floor. The slice computation starts with the set of nodes $\{req, move\}$ that directly influence the given property by modifying variables mentioned in the formula. When we compute the set of SDG nodes that is backwards reachable from the initial set of nodes, we do no longer reach all nodes, since *open* and *close* can not be reached via control or data dependencies.

Our improved slicing algorithm can therefore achieve reductions in cases where previously this was not possible. In comparison to the original specification, its slice with respect to $\Box\Diamond(aFloor = rFloor)$ does not contain methods $\mathtt{com_}open$ and $\mathtt{com_}close$ as well as variable *door*. This result is sound, since handling the elevator's door is an aspect of the original specification that does not have any influence on the given property.

6 Conclusion

This contribution proposes the combination of the well known techniques of program slicing and deductive verification. A similar combination of program slicing and constraint solving has been proposed by [Sne96] with the goal of improving the precision of data dependencies and thus more precise slices of C programs. In our case, slicing in the context of formal verification is combined with the computation of control loop invariants that are used to achieve optimised control dependencies in the dependence graph and thus a more precise slicing result. Currently, we work on the correctness proof for the proposed algorithms.

Future steps will include the extension of this approach by using methods of compositional verification. We also plan to extend this idea to the parallel composition of several CSP-OZ specifications.

References

[Brü04] I. Brückner. Slicing CSP-OZ Specifications. In *NWPT 2004*. Uppsala University, 2004.

[BW05] I. Brückner and H. Wehrheim. Slicing an Integrated Formal Method for Verification. In *ICFEM 2005*, LNCS. Springer-Verlag, 2005.

[Fis97] C. Fischer. CSP-OZ: A Combination of Object-Z and CSP. In *FMOODS'97*, volume 2, pages 423–438. Chapman & Hall, 1997.

[HDZ00] J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing Software for Model Construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990.

[NNH05] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2005.

[Smi00] G. Smith. *The Object-Z Specification Language*. Kluwer, 2000.

[Sne96] G. Snelting. Combining Slicing and Constraint Solving for Validation of Measurement Software. In *SAS*, pages 332–348, 1996.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[TS00] H. Treharne and S. Schneider. How to Drive a B Machine. In *ZB '00*, pages 188–208. Springer, 2000.