Decomposition for Compositional Verification

Björn Metzler, Heike Wehrheim, and Daniel Wonisch

Universität Paderborn
Institut für Informatik
33098 Paderborn, Germany
{bmetzler,wehrheim,dwonisch}@uni-paderborn.de

Abstract. Compositional verification is based on the idea that the correctness check of a complex system can be divided into smaller verification tasks for its components. In this paper, we show how to *decompose* a specification into components when either no such decomposition is given, or when the given composition does not lend itself to an efficient compositional verification. Our decomposition is the starting point for an application of the L* learning algorithm, generating assumptions for an assume-guarantee reasoning. We prove correctness of the decomposition as well as present experimental results using the model checker FDR2 as the teacher during learning.

1 Introduction

In formal system development verification ensures that the system meets the requirements set out by the designers or customers. Most often *model checking* is applied in the verification process to free the developer from manual proofs of correctness. Despite enormous progress made in this area ever since the invention of model checking [6], the problem of state explosion still hampers the verification of large systems. A lot of research today is still devoted to developing techniques which consequently allow model checking to scale to complex systems. Such methods range from symbolic model checking with BDDs or SAT techniques via symmetry or partial order reductions to various sorts of abstraction mechanisms.

One such technique – and the one we will be interested in here – is *compositional verification* [9]. Compositional verification takes a divide-and-conquer approach to checking correctness: instead of verifying the system as a whole, the system components are checked and the verification results are combined. One specific approach to compositional verification is *assume-guarantee (AG) reasoning* [13, 16, 19]. The verification of a system $S = S_1 \mid\mid S_2$ with respect to a property *Prop* is carried out in two steps: first, we show that S_2 guarantees *Prop* under an assumption A about its environment, and then S_1 is shown to guarantee this assumption. As a proof rule:

$$\frac{\begin{array}{c}\langle A\rangle\ S_2\ \langle \mathit{Prop}\rangle \\ \langle \mathit{true}\rangle\ S_1\ \langle A\rangle\end{array}}{\langle \mathit{true}\rangle\ S_1 \mid\mid S_2\ \langle \mathit{Prop}\rangle}$$

The drawback of this rule is the use of an assumption A which needs to be found before verification can proceed. Recently, a new technique for automatic generation of assumptions based on learning has been proposed [8]. This technique starts with a general assumption and uses a model checker as a teacher to progressively make this assumption more precise until it either matches the premises of the above proof rule or the property can be shown not to hold. The efficiency of the learning algorithm (and thus of the AG reasoning) depends on the actual decomposition of the system [7]; ideally the assumption A should be much smaller than the component S_1. Moreover, the technique relies on the *existence* of a structuring of the system into parallel components.

In this paper we will be concerned with *constructing* decompositions in case that (a) the system is not structured into parallel components, or (b) the existing structure does not lend itself to efficient assume-guarantee reasoning (e.g. because the assumption A gets too large). The starting point for our technique is a set of formal specifications written in CSP-OZ [11], a combination of the process algebra CSP [15] and the state-based formalism Object-Z [25]. The semantics of CSP-OZ are defined in terms of the semantic domain of CSP. Given a CSP-OZ specification, we first construct its *dependence graph* containing control flow as well as data dependencies among specification elements (here, Z schemas). The dependence graph construction follows a technique proposed in [5] for slicing CSP-OZ specifications. The graph is next *cut* into (currently two) parts. Roughly speaking, these two parts represent the two parallel components of the system; a definition of valid cuts and an appropriate synchronisation of the components has to ensure that the decomposition does not change the overall semantics of the specification. We consequently prove correctness of the decomposition. The cut determines the *interface* between components; by choosing a small cut we can produce small assumptions for AG reasoning.

The components we obtain through this decomposition are the starting point for the above sketched compositional verification, in which we use the technique proposed in [8] to *learn* the assumption. The employed L^* learning algorithm [1] for regular languages requires a teacher to answer membership and equivalence queries. In [8] the teacher is a model checker. As we are working in the semantic domain of CSP, we use the CSP model checker FDR2 as teacher and are thereby able to evaluate the effectiveness of the decomposition. It turns out that a compositional verification of our generated decomposition can outperform FDR2's performance during a non-compositional verification on the system as well as during a compositional verification starting with the given decomposition of the system. This is exemplified by a case study of a CSP-OZ specification of the Two-Phase-Commit Protocol and its natural – as well as generated – decomposition.

2 Background and Example

The running example for this paper on which we illustrate our decomposition as well as the verification is the Two-Phase-Commit Protocol (TPCP) [3]. We specify this protocol in CSP-OZ, an integrated formalism combining CSP and

Object-Z. While CSP is responsible for specifying the ordering of operations in the two phases of the protocol, Object-Z takes on the role of fixing what the operations themselves do.

The purpose of the protocol is to guarantee consistency of N local sites (or *pages*) of a distributed database. Instructed by a coordinator process, the protocol results in either all pages committing their transaction or all pages aborting it. As the name says the protocol works in two phases:

- *Phase 1: Commit-request*: The protocol starts with the coordinator process informing all participating pages about a *request* to commit the current transaction. Next, all pages *execute* the transaction and send a *vote* to the coordinator dependent on whether the local transaction succeeded (YES) or failed (NO). The coordinator collects the votes and *decide*s to either $COMMIT$ in the case that all votes agree on YES, or to $ABORT$ the transaction.
- *Phase 2: Commit*: The coordinator *inform*s all pages about the decision. All participating sites behave accordingly: an abort leads to an *undo* of the transaction while a commit leads to *complet*ion. In any case, the sites output the *result* and send an *acknowledge*ment to the coordinator.

Let N be the number of pages participating in the protocol and let $Votes$ and $Trans$ be the following two base types:

$Votes == \{YES, NO\}$
$Trans == \{COMMIT, ABORT\}$

The specification given below is the CSP-OZ class for the central coordinator.

Coord

$\mathtt{method}\ request, vote[x? : Votes], decide, inform[x! : Trans], acknowledge$

$\mathtt{main} \overset{c}{=} |||_{0<i\leq N}(request \rightarrow \mathtt{Skip});\ |||_{0<i\leq N}(vote \rightarrow \mathtt{Skip});$
$\quad decide \rightarrow |||_{0<i\leq N}(inform \rightarrow \mathtt{Skip});\ |||_{0<i\leq N}(acknowledge \rightarrow \mathtt{Skip})$

$dec : Trans$
$votes : \mathbb{P}\ Votes$

Init
$dec = ABORT$
$votes = \varnothing$

request
$\Delta(votes)$
$votes' = \varnothing$

acknowledge

vote
$\Delta(votes);\ vo? : Votes$
$votes' = votes \cup \{vo?\}$

inform
$in! : Trans$
$in! = dec$

decide
$\Delta(dec)$
$\mathbf{if}\ (NO \in votes)\ \mathbf{then}\ dec' = ABORT\ \mathbf{else}\ dec' = COMMIT$

The first part of the class defines its interface, i.e. the methods/operations (or channels in the CSP terminology) with their signatures. The next part is a CSP process equation describing the ordering of operations of the coordinator. Class *Coord* starts by sending a request to all N pages, accepts all the votes, decides, and consequently informs all pages about the decision, and finally waits for an acknowledge. This ordering is specified using the CSP operators for interleaving ($|||$) – a special form of parallel composition – sequencing ($;$), prefixing of operations ($\rightarrow$) and the empty terminating process *Skip*. The third part – the Object-Z part – consists of a number of schemas specifying the class' state space, initialisation and the operations. The class has two variables: *dec* for holding the final decision and a set of votes *votes*. The operations can or cannot modify these variables (specified by the Δ-list) and input and output variables (marked ? and !, respectively) help to pass values between classes. For instance, operation *vote* stores an input *vo*? in the variable *votes*. Input and output variables are in general not restricted by the CSP part. Therefore, we refrain from denoting them there.

Note that an empty schema describes an operation which leaves all variable values unchanged. In the following we will leave out empty schemas. Both parts, CSP as well as Object-Z, impose restrictions on the behaviour of the class which need to be jointly obeyed.

The class *Coord* operates in parallel with several instantiations of the following class *Page*.

Page

$\texttt{method}\ \mathit{request}, \mathit{execute}, \mathit{vote}[x! : \mathit{Votes}], \mathit{inform}[x? : \mathit{Trans}]$

$\texttt{method}\ \mathit{undo}, \mathit{complete}, \mathit{result}[b! : \mathit{Trans}], \mathit{acknowledge}$

$\texttt{main} \overset{c}{=} \mathit{request} \rightarrow \mathit{execute} \rightarrow \mathit{vote} \rightarrow \mathit{inform} \rightarrow P$

$P \overset{c}{=} \mathit{undo} \rightarrow \mathit{result} \rightarrow \mathit{acknowledge} \rightarrow \texttt{Skip}$

$\quad \Box\ \mathit{complete} \rightarrow \mathit{result} \rightarrow \mathit{acknowledge} \rightarrow \texttt{Skip}$

$\mathit{dec} : \mathit{Trans}$
$\mathit{stable} : \mathbb{B}$

Init
$\mathit{dec} = \mathit{ABORT}$
stable

result
$b! : \mathit{Trans}$
$b! = \mathit{dec}$

execute
$\Delta(\mathit{stable})$
$\mathit{stable}' \in \{\mathrm{true}, \mathrm{false}\}$

undo
$\mathit{dec} = \mathit{ABORT}$

complete
$\mathit{dec} = \mathit{COMMIT}$

inform
$\Delta(\mathit{dec})$
$\mathit{in}? : \mathit{Trans}$
$\mathit{dec}' = \mathit{in}?$

vote
$\mathit{vo}! : \mathit{Votes}$
$\mathit{stable} \Rightarrow \mathit{vo}! = \mathit{YES}$
$\neg \mathit{stable} \Rightarrow \mathit{vo}! = \mathit{NO}$

Here, we employ an additional CSP operator: the choice operator ($\Box$) for choosing between alternatives. The choice between *undo* and *complete* is determined by the Object-Z part: *undo* has a predicate $dec = ABORT$ in its schema (which works as a precondition), while *commit* can only be executed when *dec* equals *COMMIT*.

The full system is specified as

$$System = Coord \|_S (|||_{0<i\leq N} Page)$$

with $S = \{request, vote, inform, acknowledge\}$. Here, we use a different CSP parallel composition operator: $\|_S$ requires joint execution of the operations (or events in CSP terminology) in S, i.e. *Coord* and *Pages* need to synchronize on *request*, *vote*, *inform* and *acknowledge*. For the remainder of this paper, let $Pages = |||_{0<i\leq N} Page$. This completes the specification of the TPCP.

Next, we are interested in verifying a specific property of the Two-Phase-Commit protocol, i.e. of *System*. The property is a safety property and states that if at least one page votes *NO*, all pages will *undo* the transaction. Before formally specifying this property, we first have to clarify the language we use for writing properties. The formalism CSP-OZ has a joint semantics for CSP and Object-Z parts, which is given in terms of CSP alone. CSP on the other hand has a semantics given within the *failures-divergences* model. Here, we will solely be interested in safety properties and move to the simpler domain of *traces*. Traces represent the behaviour of a system in terms of the possible orderings of its events. Thus the basis for our semantics is a set *Events* representing all valid events of our system. This set consists of operation names together with values for parameters. For our example, *Events* contains the events *inform.ABORT*, *decide*, *vote.NO*, etc. Given this set *Events*, the trace semantics of a system S is a (prefix-closed) set of traces:

$$traces(S) \subseteq 2^{Events^*}$$

As with the systems, we also specify properties as sets of traces, namely simply by giving all valid traces possessing a particular property. While we could also use CSP-OZ for property specification, here we will stick to CSP for this purpose. The following CSP specification presents our correctness property for the TPCP:

$$\begin{aligned}
PROP &= PC(N)\\
PC(0) &= |||_{0<i\leq N}\, complete \rightarrow \texttt{Skip}\\
PC(j) &= vote.YES \rightarrow PC(j-1) \,\Box\, vote.NO \rightarrow PU(j-1)\\
PU(0) &= |||_{0<i\leq N}\, undo \rightarrow \texttt{Skip}\\
PU(j) &= \Box_{x:\{YES,NO\}}\, vote.x \rightarrow PU(j-1)
\end{aligned}$$

The process $PC(j)$ allows for j votes (and thus *PROP* for N votes) and - if control flow has not left the process before - finally N events *complete*. As soon as one vote is *NO*, PC processes switch to some process PU. $PU(j)$ also allows for j votes (with any parameter value) but always terminates with *undo*s. Thus as soon as we have one event *vote.NO*, the final events of *PROP* will be *undo*. The question is now whether the traces of *System* (concerning the interesting events *vote*, *undo* and *complete*, here extracted by hiding (\) all other events) are contained in those of *PROP*, i.e.

$$\begin{aligned} &traces(System \backslash \{request, execute, inform, decide, result, acknowledge\}) \\ &\qquad \subseteq traces(PROP) \end{aligned}$$

This check for trace inclusion is a standard check for CSP specifications as one of CSP's refinement orderings is trace inclusion.

Definition 1. *Let P, Q be CSP processes. P is a* trace refinement *of Q, if $traces(P) \subseteq traces(Q)$. We write $Q \sqsubseteq_T P$. P is* trace equivalent *to Q, $P =_T Q$, if $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$.*

This being standard for CSP, we can use the CSP model checker FDR2 [18] to check it, using a technique proposed in [12] to translate a CSP-OZ specification into CSP. As it turns out, FDR2 fails to carry out this check for more than 5 pages.

Next, we tried to use assume-guarantee reasoning to show the property. The *System* specification is already structured using parallelism, with *Coord* being S_1 and all *Pages* S_2. Thus we used the AG rule given in the introduction. This rule has been proven correct and complete [21]. Rephrased in terms of CSP's traces refinement, the rule reads:

$$\frac{\begin{array}{c} PROP \sqsubseteq_T A \mid\mid_X S_2 \\ A \sqsubseteq_T S_1 \end{array}}{PROP \sqsubseteq_T S_1 \mid\mid_Y S_2} \tag{1}$$

Here, the sets X and Y are synchronisation sets defined by $X = \alpha(A) \cap \alpha(S_2)$ and $Y = \alpha(S_1) \cap \alpha(S_2)$ (intersection of alphabets). The assumption A represents a restriction on S_2's environment which is necessary for S_2 to guarantee *PROP*. On the other hand, S_1 needs to guarantee this restriction. For generating the assumption we use the technique proposed in [8]. This method employs Angluin's L* algorithm for learning regular languages (a finite automaton) to generate the assumption A. The algorithm needs to employ a teacher which can answer membership as well as equivalence queries. The proposal of [8] was to use a model checker to this end. We have a prototypical implementation of the L* algorithm which takes CSP processes as inputs, calls the CSP model checker FDR2 whenever a teacher is required and ultimately either outputs 'true' and an assumption if $PROP \sqsubseteq_T S_1 \mid\mid_Y S_2$ is true or 'false', if not. Using the given structure of *System*, the assume-guarantee reasoning unfortunately gives us no gain at all. In the contrary, the run-times get worse and the check fails as soon as we reach 5 pages (see the section on experimental results). The reason for this is that the part which produces most of the complexity is *Pages* as it is the interleaving of a large number of processes. The state space of *Pages* needs to be constructed in both the verification of the complete system and in the

AG verification. It would be preferable to have a decomposition of the system which *splits* the process *Pages* such that the compositional verification never needs to consider *Pages* in its entirety. Next, we will see how to construct such a decomposition of the system.

3 Decomposition of a Specification

We aim to decompose a specification into two components allowing the application of assume-guarantee reasoning. To find a suitable decomposition in this context we need to analyse a specification's *dependence structure*: the specification's elements (operations of a class) might depend on each other. The distribution of dependent elements over both components is not desirable. However, if required, it necessitates that an assumption describes the correlation between the different operations. Thus we need to define what *dependence* means and we need to ensure that the decomposition preserves the overall dependence structure of the specification. To find a small assumption – and this is preferable – the number of intersecting dependencies between the components should be small.

Fortunately, for CSP-OZ and in the context of program slicing [28], Brueckner [4] developed a precise dependence analysis for CSP-OZ and defined a specification's *Dependence Graph*:

Definition 2. *(Dependence Graph of a specification)*
The Dependence Graph *(DG)* $G = (N, \rightarrow_{DG})$ *of a specification* S *is defined over a set of nodes* $N = cf(N) \cup op(N)$ *and a set of edges* $\rightarrow_{DG} = \rightarrow \cup \dashrightarrow$*. The set* $cf(N)$ *corresponds to operators within the specification's CSP part whereas* $op(N)$ *corresponds to operations of the specification.*[1] *For the set of edges, we distinguish control flow edges (*$\rightarrow$*) from program dependence edges (*$\dashrightarrow$*).*

The set of DG edges describes dependencies between different nodes of the DG mostly following the principle of cause and effect – i.e. the edge's source node controls or influences execution of its target node. The DG comprises the *Control Flow Graph (CFG)* and the *Program Dependence Graph (PDG)*.

As the name says, the CFG covers dependencies with respect to the specification's control flow structure. This is mainly derived from its CSP part. As an example: in class *Page*, *request* prefixes *execute* leading to a control flow edge from *request* to *execute*.

Along with this, the PDG edges describe dependencies between different operations of the specification such as data-, control-, synchronisation data- or interference data dependencies. They refer to the set of state variables of the class' Object-Z part. An example of a data dependence is the edge from *execute* to *vote* – the variable *stable* is modified within *execute* and referenced in *vote*, i.e. $stable \in mod(execute) \cap ref(vote)$ where $mod(op)$ and $ref(op)$ denote the

[1] For simplicity, instead of defining the DG with respect to predicate nodes, we use operation nodes.

sets of *modified* and *referenced* variables within an operation *op*, respectively. A synchronisation data dependence exists between the events *execute* of both classes since *Page.execute* has an output that *Coord.execute* uses as an input. Note that PDG edges only connect operation nodes.

As a multiple occurrence of an Object-Z operation within the CSP part of a specification is possible, we define the correlation between operation nodes of the DG and operations of the specification:

Definition 3. *(Labelling of DG nodes)*
Let $G = (N, \rightarrow_{DG})$ be the DG of a specification S and let Op be the set of all operations of S. The labelling function $l : op(N) \rightarrow Op$ *maps an operation node of the DG on its corresponding operation name within S. For $O \subseteq Op$, we define $l^{-1}[O] := \{n \in op(N) \mid l(op) \in O\}$.*

In the following, we assume the alphabet of the CSP part and the set of Object-Z operations to be equal. Thus, for the CSP part, *Op* is the set of events projected on its *names* omitted from its parameters.

For a complete definition of a specification's DG, see [4]. Figure 1 shows (a slightly simplified version of[2]) the DG for *System*. We use different types of arrows to illustrate control flow- ($\rightarrow$) and program dependence-edges ($\dashrightarrow$).

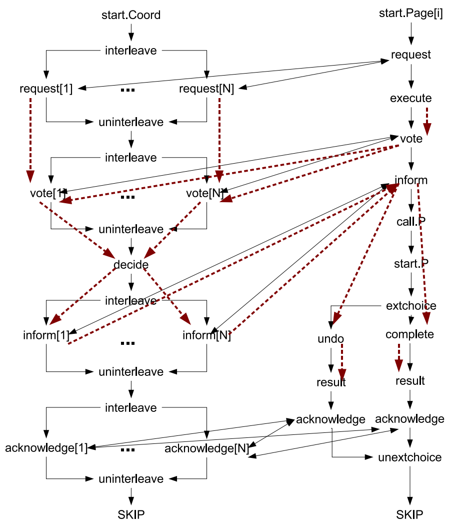

Fig. 1. Dependence Graph for *System*

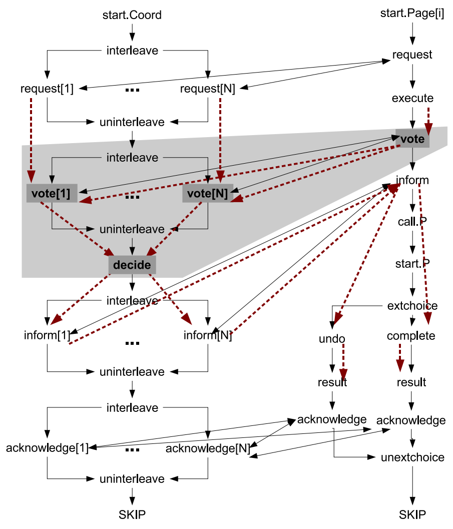

Fig. 2. Cut for $l^{-1}[\{decide, vote\}]$

[2] Here, we omit the additional *start*- and *term*-nodes for the parallel composition of both classes. We do also not incorporate the intermediate *seq*-nodes within *Coord* for the sequential composition of the interleavings. For an operation *op*, we use $op[i]$ to depict its i-th execution within the corresponding CSP-interleaving. $Page[i]$ is an arbitrary instance of *Page*.

3.1 A Cut of a Dependence Graph

All dependencies between different specification elements are represented in the specification's DG. Thus to decompose a given specification S into two parts, we start by defining a decomposition of the DG.

The basic idea is the definition of a *cut* $\mathbf{C}$ identifying the interface between the parallel components which we define subsequently. Being a subset of the DG's operation nodes, a cut fragments the DG into two subgraphs representing the two stages (phases) of the graph. The cut will then yield a decomposition of the specification itself. In the context of assume-guarantee reasoning, we set the following objectives:

- The overall semantics of S are preserved, i.e. the original specification is trace equivalent to its decomposition, when both parts are combined via parallel composition,
- the decomposition is efficient in the context of assume-guarantee reasoning, i.e. a cut leads to a relatively small intersection between the components and uniformly distributed operations.

In the remainder of the paper, we will not generally distinguish between control flow edges and program dependence edges. Next, we define the cut of a specification's dependence graph.

Definition 4
Let $G = (N, \rightarrow)$ be a graph and $N' \subseteq N$. Then,

$N'\!\downarrow := \{n \in N \mid \exists\, n' \in N' \bullet n \rightarrow^* n'\}$ *(all nodes reaching N'),*
$N'\!\uparrow := \{n \in N \mid \exists\, n' \in N' \bullet n' \rightarrow^* n\}$ *(all nodes reachable from N').*

Definition 5. *(Cut of the DG)*
Let $G = (N, \rightarrow_{DG})$ be the DG of a given specification. A cut $\mathbf{C} \subseteq op(N)$ *of G is a subset of the operation nodes of N such that*

a) $\mathbf{C}\!\downarrow \cup\, \mathbf{C}\!\uparrow = N$,
b) $\nexists c \in \mathbf{C}\!\downarrow \setminus \mathbf{C}, c' \in \mathbf{C}\!\uparrow \setminus \mathbf{C} \bullet c \rightarrow_{DG} c' \vee c' \rightarrow_{DG} c$,
c) $\forall\, n_1, n_2 \in op(N) \bullet l(n_1) = l(n_2) \Rightarrow (n_1 \in \mathbf{C} \Leftrightarrow n_2 \in \mathbf{C})$

$\mathbf{C}$ is a subset of the DG's operation nodes determining a split of the DG into $\mathbf{C}\!\downarrow$ and $\mathbf{C}\!\uparrow$ and it is used to define the decomposition of a specification S into S_1 and S_2. S_1 and S_2 are the parallel components of the decomposition with the intersection of $\mathbf{C}\!\downarrow$ and $\mathbf{C}\!\uparrow$ defining the interface between S_1 and S_2.

Condition a) states that for any node $n \in N$, either the cut is reachable from n or n is reachable from the cut. Therefore, no node will be left out. *Condition b)* states that DG edges must not cross the cut. This condition ensures that there are no dependencies from one component to another circumventing the cut. *Condition c)* states that for operations with multiple occurrence within the CSP part, either *all* or *none* of the corresponding DG nodes are contained in the cut. This condition is required to ensure proper synchronisation between the constructed parallel components.

In the context of a cut $\mathbf{C}$, we will call $\mathbf{C}{\downarrow}$ the *precut* and $\mathbf{C}{\uparrow}$ the *postcut*. Next, we define a condition on the relation between $\mathbf{C}{\downarrow}$ and $\mathbf{C}{\uparrow}$ to restrict the distribution of a DG:

Definition 6. *(sequential cut)*
Let $G = (N, \rightarrow_{DG})$ be the DG of a given specification. A cut $\mathbf{C}$ ***sequentially distributes*** *G, iff $\mathbf{C}{\uparrow} \cap \mathbf{C}{\downarrow} = \mathbf{C}$. We call $\mathbf{C}$ a* sequential cut.

The condition for the sequential distribution of a DG states that there are no nodes leading to the cut which are also reachable from the cut. Thus, the DG can be viewed in two stages, with a unique distribution of all nodes: a first stage before the cut and a second stage starting from the cut, with the cut itself being their intersection. In particular, a sequential cut requires that all cycles of the DG are distributed over the resulting two subgraphs without intersecting with the cut itself. All paths connecting both subgraphs must pass the cut.

Even though we consider a rather specific, simple class of dependence graphs to illustrate our approach and the applicability of the assume-guarantee proof rule, our approach is not restricted to sequential distributions: the definition of a cut can as well be applied to circular dependence graphs. This will be part of our future work.

For our example, based on two heuristics for the definition of a cut, the decomposition of the DG with respect to the set $\mathbf{C} = l^{-1}[\{vote, decide\}]$[3] is given in Figure 2. These heuristics can informally be described as follows:[4]

- A cut should contain as few as possible nodes and its corresponding operations should modify as few as possible variables,
- A cut should be defined in the middle part of the DG.

Using the set $\mathbf{C}' = \{decide\}$ would lead to a violation of the cut definition due to the cut-crossing CFG edge from *vote* to *inform* on the right hand side of the DG. Therefore, we additionally needed to add *vote*. $\{decide, vote\}$ indeed defines a sequential cut since neither there are cut-crossing edges nor nodes outside of $\mathbf{C}$ assigned to the first and to the second stage. Condition c) holds as well since all DG nodes assigned to the operation *vote* are contained in the cut.

3.2 Decomposition of a Specification

As a next step, we define the decomposition of a specification. This will be done with respect to a sequential cut of its DG. Precut and postcut will be used to define two components S_1 and S_2 with the following goal: $S_1 \parallel S_2$ has the same set of traces as S and is therefore – in our semantic domain – equivalent to S. The decisive point in this definition is the synchronisation alphabet: we need to guarantee correct values for the state variables in the second stage. These variables might have been modified during the first stage. Synchronization should

[3] In the following cut-examples, we will synonymously use S and $l^{-1}[S]$.
[4] A closer investigation of heuristics for selecting optimal cuts will form part of our future work.

thus lead to a passing of the current values. To ensure this, we use the set of cut events as the synchronisation alphabet and identify all variables modified during the first stage. These are then communicated to the second stage.

A CSP-OZ definition of a class S consists of the following elements:

S

I	[interface definition]
`main`	[CSP part]
State	[OZ part: state schema]
Init	[OZ part: initial state schema]
op	[OZ part: operations]

For $m \in Op$, a method declaration has the form $m[p_1 : t_1, \dots, p_m : t_m]$ with parameters p_i of type t_i. For the corresponding Object-Z operation, *op.par* denotes its parameter declaration and *op.pred* its predicate part. *Var* denotes the set of state variables of a class. For $M \subseteq Op$, let $\{\mid M \mid\} := \{m.i.o \in Events \mid m \in M\}$.

To define the decomposition, we first need to define a projection of a CSP process to a subset of its events. This projection will then be used to decompose the CSP part with respect to the precut and the postcut.

Definition 7. *(Projection of CSP processes, [4])*
Let P be the right-hand side of a CSP process definition and $E \subseteq Events$. The projection of P on E*, denoted by $P|_E$, is inductively defined:*

1. $\texttt{Skip}|_E := \texttt{Skip}$ *and* $\texttt{Stop}|_E := \texttt{Stop}$,
2. $(e \rightarrow P)|_E := \begin{cases} P|_E, & e \notin E \\ e \rightarrow P|_E, & otherwise, \end{cases}$
3. $(P \circ Q)|_E := (P|_E) \circ (Q|_E)$ *for* $\circ \in \{;\ , |||, \Box, \sqcap\}$,
4. $(P \parallel_S Q)|_E := (P|_E) \parallel_{S \cap E} (Q|_E)$.

To determine the projection of a complete CSP part, Definition 7 has to be applied to every CSP process definition. Next, we define the decomposition of S with respect to a sequential cut:

Definition 8. *(Decomposition with respect to a sequential Cut)*
Let S be a specification and $G = (N, \rightarrow_{DG})$ be its dependence graph. Let $\mathbf{C}$ be a sequential cut and let $M_1 := l[\mathbf{C}\!\downarrow \cap op(N)]$, $M_2 := l[\mathbf{C}\!\uparrow \cap op(N)]$, $M_{\mathbf{C}} := M_1 \cap M_2$, $E_i := \{\mid M_i \mid\}$, $E_{\mathbf{C}} := \{\mid M_{\mathbf{C}} \mid\}$, $V_1 := Var(M_1)$, $V_2 := Var(M_2)$ and

$$V_{\mathbf{C}} = \{x \in S.Var \mid \exists\, n \in M_{\mathbf{C}}, n' \in (M_2 \setminus M_{\mathbf{C}}) \bullet n \rightarrow^*_{DG} n' \wedge x \in (mod(n) \cap ref(n'))\}.$$

Given a set $V_{\mathbf{C}} = \{x_1, \dots, x_n\}$ of types s_i, for $m \in Op$, let $\{x_{m_1}, \dots, x_{m_k}\} = V_{\mathbf{C}} \cap mod(m)$. We use a function f to define the interface extension of the class:

$$f(m[p_1 : t_1, \dots, p_m : t_m] =$$
$$\begin{cases} m[p_1 : t_1, \dots, p_m : t_m, a_{m_1} : s_{m_1}, \dots, a_{m_k} : s_{m_k}, b_{m_1} : r_{m_1}, \dots, b_{m_l} : r_{m_l}], & m \in M_{\mathbf{C}} \\ m[p_1 : t_1, \dots, p_m : t_m], & otherwise \end{cases}$$

The decomposition of S with respect to $\mathbf{C}$ into S_1 and S_2 *is defined as*[5]*:*

S_1

$I_1 := f(I|_{M_1})$

$\texttt{main}_1 := \texttt{main}|_{E_1}$ [extended by additional parameters]

$\texttt{State}_1 := \texttt{State} \upharpoonright V_1$

$\texttt{Init}_1 := \texttt{Init} \upharpoonright V_1\ (*)$

$$op_1 := \begin{cases} op, & op \in M_1 \setminus M_{\mathbf{C}} \\ [op.par, a_{m_i}! : s_{m_i}, b_{m_j}? : r_{m_j} \mid op.pred \wedge \bigwedge_{i=1}^{k} a_{m_i}! = x'_{m_i}], & op \in M_{\mathbf{C}} \end{cases}$$

S_2

$I_2 := f(I|_{M_2})$

$\texttt{main}_2 := \texttt{main}|_{E_2}$ [extended by additional parameters]

$\texttt{State}_2 := \texttt{State} \upharpoonright V_2$

$\texttt{Init}_2 := \texttt{Init} \upharpoonright V_2\ (*)$

$$op_2 := \begin{cases} op, & op \in M_2 \setminus M_{\mathbf{C}} \\ [op.par, a_{m_i}? : s_{m_i}, b_{m_j}? : r_{m_j} \mid \bigwedge_{i=1}^{k} x'_{m_i} = a_{m_i}?], & op \in M_{\mathbf{C}} \end{cases}$$

$I|_M$ depicts the restriction of the set of methods within the interface I onto M. For the initial state schemas $\texttt{Init}_i$, the definition is annotated with an asterisk: the coarse idea is a projection of `Init` onto all predicates solely dealing with V_i. Atomic predicates sharing variables local to S_1 and S_2 need to be restricted but can not be left out. Here, we omit a detailed definition of $\texttt{Init}_i$.

We take a closer look at the additional parameters for $m \in M_{\mathbf{C}}$. Firstly, $V_{\mathbf{C}}$ defines exactly the set of state variables from the first stage that influence the second stage of the specification. Note that any such variable must be modified *inside* some cut event since otherwise there would be cut-crossing edges in the DG. For each $x_i \in V_{\mathbf{C}} \cap mod(op)$, we use parameters $a_{m_i} : s_{m_i}$ extending the type of op. These parameters uncover the influence of the first component on the second one. Secondly, Definition 5 allows for a cut containing two or more DG nodes with the same labelling. Since parallel composition based on events does not distinguish between these nodes, we need to ensure that in the decomposition, corresponding instances of the event are synchronized. This is achieved by adding additional *address parameters* $b_{m_j} : r_{m_j}$ to the respective channels solely being restricted by the CSP part. On the one hand, these parameters ensure that all previously allowed synchronisations are still possible. On the other hand, synchronisation between S_1 and S_2 is restricted to matching DG nodes. The number of address parameters depends on the number of classes synchronizing on op whereas the type r_{m_j} depends on the cardinal number of $l^{-1}[\{op\}]$. We will exemplify this on our example and refrain from giving a precise definition here.

For the Object-Z part, we extend any operation of the cut with corresponding additional outputs (S_1) and inputs (S_2), respectively. Moreover, we eliminate the remaining predicate part of the shared operations within S_2.

[5] `State` $\upharpoonright V$ denotes the projection of *State* on a subset V of its state variables.

3.3 Example Revisited

We apply the decomposition to our example. Based on our objective to decompose a specification into equally distributed components, we define several sequential cuts and investigate each one. The verification results are given in the next section. Here, we illustrate the decomposition for the set $E_{\mathbf{C}} = \{vote, decide\}$ as introduced in Section 3.1. We split E into

$E_1 = \{request, execute, vote, decide\}$ and
$E_2 = \{vote, decide, inform, undo, complete, failure, result, acknowledge\}$.

The definition of a DG's cut is not restricted to a single class. In fact, the DG is defined with respect to the full specification possibly containing several classes. Since *System* comprises two classes *Coord* and *Page*, both will be decomposed into components $Coord_1, Coord_2$ and $Page_1, Page_2$, respectively. For the parallel composition $System_1 \;\|_{E_{\mathbf{C}}}\; System_2$, $System_i$ is defined as $Coord_i \;\|\; Pages_i$. $System_1$ can be viewed as the first and $System_2$ as the second stage of the protocol. Control flow according to *System* is restored by the parallel composition.

According to Definition 8, the resulting specification slices are given below. Since $V_{\mathbf{C}} = \{Coord.dec\}$, event *decide* is extended by one parameter for the communication of *Coord.dec*. Also, event *vote* is extended by two additional parameters to ensure synchronisation for matching occurrences of *vote*. We explicitly depict these parameters in the CSP part since they are restricted there. Based on $\mid l^{-1}[\{Coord.vote\}] \mid = \mid l^{-1}[\{Pages.vote\}] \mid = N$, these parameters are of type $\mathbf{N} = \{1, \ldots, N\}$. To address specific instances of $Page_1$ and $Page_2$, we adopt CSP-OZ's concept of constant parameters, and use $Pages_j = |||_{0<i\leq N} Page_j(i)$.

$Coord_1$

`method` $request, vote[p_1? : \mathbf{N};\; p_2? : \mathbf{N};\; x? : Votes], decide[a! : Trans]$

`main` $= |||_{0<i\leq N}(request \rightarrow \texttt{Skip});\; |||_{0<i\leq N}(vote?p_1.i \rightarrow \texttt{Skip});\; decide \rightarrow \texttt{Skip}$

$dec : Trans$
$votes : \mathbb{P}\; Votes$

`Init`
$dec = ABORT$
$votes = \varnothing$

request
$votes' = \varnothing$

vote
$\Delta(votes)$
$p_1? : \mathbf{N};\; p_2? : \mathbf{N};\; vo? : Votes$
$votes' = votes \cup \{vo?\}$

decide
$\Delta(dec);\; a! : Trans$
$a! = dec'$
if $(NO \in votes)$ **then** $dec' = ABORT$ **else** $dec' = COMMIT$

$Page_1(i : \mathbf{N})$

$\texttt{method}\ request,\ vote[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x! : Votes],\ execute$

$\texttt{main} = request \rightarrow execute \rightarrow vote.i?p_2 \rightarrow \texttt{Skip}$

$stable : \mathbb{B}$

$\texttt{Init}$
$stable$

$vote$
$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo! : Votes$
$stable \Rightarrow vo! = YES$
$\neg stable \Rightarrow vo! = NO$

$execute$
$\Delta(stable)$
$stable' \in \{\mathrm{true}, \mathrm{false}\}$

$Coord_2$

$\texttt{method}\ vote[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x? : Votes],\ decide[a? : Trans]$
$\texttt{method}\ inform[x! : Trans],\ acknowledge$

$\texttt{main} = |||_{0<i\leq N}(vote?p_1.i \rightarrow \texttt{Skip});\ decide \rightarrow |||_{0<i\leq N}(inform \rightarrow \texttt{Skip});$
$\qquad |||_{0<i\leq N}(acknowledge \rightarrow \texttt{Skip})$

$dec : Trans$

$\texttt{Init}$
$dec = ABORT$

$inform$
$in! : Trans$
$in! = dec$

$decide$
$\Delta(dec);\ a? : Trans$
$dec' = a?$

$vote$
$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo? : Votes$

$Page_2(i : \mathbf{N})$

$\texttt{method}\ vote[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x! : Votes],\ inform[x? : Trans],\ undo$
$\texttt{method}\ complete,\ result[b! : Trans],\ acknowledge$

$\texttt{main} = vote.i?p_2 \rightarrow inform \rightarrow P$
$P = undo \rightarrow result \rightarrow acknowledge \rightarrow \texttt{Skip}$
$\qquad \Box\ complete \rightarrow result \rightarrow acknowledge \rightarrow \texttt{Skip}$

$dec : Trans$

$\texttt{Init}$
$dec = ABORT$

$inform$
$\Delta(dec);\ in? : Trans$
$dec' = in?$

$result$
$b! : Trans$
$b! = dec$

$undo$
$dec = ABORT$

$complete$
$dec = COMMIT$

$vote$
$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo! : Votes$

In [9], the motivation for introducing and specifying the TPCP is its particular structure allowing for an appliance of the *Communication-Closed-Layers law* (CCL) [10]. Our way of decomposing a specification is one particular way of adopting the CCL.

3.4 Correctness of the Decomposition

We will now show that the full specification is trace equivalent to the composition of both components constructed in Definition 8. As mentioned in Section 2, for our main goal we want to apply the assume-guarantee rule 1 from Section 2 to show $PROP \sqsubseteq_T S$. To show correctness of the decomposition we have to show

$$S =_T S_1 \,||_{E_\mathbf{C}}\, S_2, \tag{2}$$

i.e. our original specification is trace equivalent to the parallel composition of the components. Then we can apply the given rule with respect to S:

$$\frac{\begin{array}{c} PROP \sqsubseteq_T A \,||_X\, S_2 \\ A \sqsubseteq_T S_1 \end{array}}{PROP \sqsubseteq_T S}$$

The following lemma will be applied to establish the overall correlation between the specification and the decomposition:

Lemma 1
Let P_i, Q_i be CSP processes and A_i, B_i their respective alphabets. Then,

$$(P_1 \;{}_{A_1}||_{A_2}\; P_2) \;{}_{A_1 \cup A_2}||_{B_1 \cup B_2}\; (Q_1 \;{}_{B_1}||_{B_2}\; Q_2) =$$
$$(P_1 \;{}_{A_1}||_{B_1}\; Q_1) \;{}_{A_1 \cup B_1}||_{A_2 \cup B_2}\; (P_2 \;{}_{A_2}||_{B_2}\; Q_2)$$

Proof
We use rule (2.5)

$$(P \,||_{X \cap Y}\, Q) \,||_{(X \cup Y) \cap Z}\, R = P \,||_{X \cap (Y \cup Z)}\, (Q \,||_{Y \cap Z}\, R)$$

from [23], p. 57 and incrementally deduce the equation. □

Next, we state the main theorem of this paper: the decomposition of a specification based on a sequential cut is trace equivalent to the original specification.

Theorem 1. *(Correctness of the Decomposition)*
Let S be a specification and $G = (N, \rightarrow_{DG})$ be its DG. Let $\mathbf{C}$ be a sequential cut and let S_1 and S_2 be the decomposition of S with respect to Definition 8. Then, the following holds:

$$S =_T S_1 \,||_{E_\mathbf{C}}\, S_2 \tag{3}$$

Proof
In our semantic domain we are interested in $traces(S) \subseteq 2^{Events^*}$. Based on the CSP trace semantics for CSP-OZ we get $traces(S) := traces(\texttt{main} \,||_{Events}\, OZ)$. Thus, a trace of a CSP-OZ class is a trace within the parallel composition of the specification's CSP part and Object-Z part, respectively, synchronizing on the set *Events*. We compositionally show (3) by dealing with the specification's CSP- and Object-Z part independently. If we can show

$$\texttt{main} =_T \texttt{main}_1 \,||_{E_\mathbf{C}}\, \texttt{main}_2, \tag{4}$$

$$OZ =_T OZ_1 \,||_{E_\mathbf{C}}\, OZ_2 \text{ for the set of traces of the CSP part,} \tag{5}$$

we can subsequently apply Lemma 1 with respect to $A_1 = A_2 = E_1$, $B_1 = B_2 = E_2$ and deduce

$$\begin{array}{ll}
\quad traces(S) & \\
= traces(\texttt{main} \,||_{Events}\, OZ) & (Def.\ of\ S) \\
= traces((\texttt{main}_1 \,||_{E_\mathbf{C}}\, \texttt{main}_2) \,||_{Events}\, (OZ_1 \,||_{E_\mathbf{C}}\, OZ_2)) & ((4), (5)) \\
= traces((\texttt{main}_1 \,||_{E_1 \cap E_2}\, \texttt{main}_2) \,||_{Events}\, (OZ_1 \,||_{E_1 \cap E_2}\, OZ_2)) & (E_1 \cap E_2 = E_\mathbf{C}) \\
= traces((\texttt{main}_1 \,||_{E_1}\, OZ_1) \,||_{E_1 \cap E_2}\, (\texttt{main}_2 \,||_{E_2}\, OZ_2)) & (Lemma\ 1) \\
= traces((\texttt{main}_1 \,||_{E_1}\, OZ_1) \,||_{E_\mathbf{C}}\, (\texttt{main}_2 \,||_{E_2}\, OZ_2)) & (E_1 \cap E_2 = E_\mathbf{C}) \\
= traces(S_1 \,||_{E_\mathbf{C}}\, S_2) & (Def.\ of\ S_1, S_2)
\end{array}$$

Due to lack of space, we refrain from giving the complete proofs of (4) and (5) but outline the ideas. The core idea for (4) is to assume that any trace $tr \in traces(\texttt{main})$ has the following structure:

$$\underset{\boxed{E_1}}{tr_1} \;{}^\frown\; \underset{\boxed{E_C}}{tr_\mathbf{C}} \;{}^\frown\; \underset{\boxed{E_2}}{tr_2}$$

We then show that $tr \in traces(\texttt{main})$ if and only if $tr_1 {}^\frown tr_\mathbf{C} \in traces(\texttt{main}_1)$ and $tr_\mathbf{C} {}^\frown tr_2 \in traces(\texttt{main}_2)$ holds. Here, we particularly use Condition c) of Definition 5. However, if tr switches between different interleaving branches of the CSP part, an event of $E_2 \setminus E_\mathbf{C}$ can be executed before an event of $E_1 \setminus E_\mathbf{C}$ without violating the cut definition. To solve this problem, we apply Lemma 1 to restructure the trace and treat interleaving branches separately.

For the Object-Z part, $OZ =_T OZ_1 \,||_{E_\mathbf{C}}\, OZ_2$ would be preferable. However, this equivalence does in general not hold: if the CSP part does not determine the ordering of events, a trace within $traces(OZ)$ may not correspond to the paths of the DG. The cut is defined with respect to the DG, the decomposition might access and output inconsistent values. Thus, $tr \in traces(OZ)$ does not need to be an element of $traces(OZ_1 \,||_{E_\mathbf{C}}\, OZ_2)$ and vice versa. Indeed, we show the following, weaker property for the Object-Z part:

$$\forall\, tr \restriction Op \in traces(\texttt{main}) \restriction Op \bullet tr \in traces(OZ) \Leftrightarrow tr \in traces(OZ_1 \,||_{E_\mathbf{C}}\, OZ_2) \tag{6}$$

It states that $OZ =_T OZ_1 \,||_{E_\mathbf{C}}\, OZ_2$ if the CSP part determines the ordering of events within the trace. Since this will always be the case for a trace within a CSP-OZ specification, it is sufficient to show (5). To do so, given $tr \in traces(OZ)$, we need to construct tr_i $traces(OZ_i)$ such that both synchronize on $E_\mathbf{C}$. For the reverse direction, we have to construct an equivalent $tr \in traces(OZ)$ out of $tr_i \in traces(OZ_i)$.

The complete proof of (4) and (5) uses all the various dependencies of the PDG. For instance, data dependencies ensure that in case a certain variable is modified, it always refers to the correct variable values used in the modification. Synchronization dependencies ensure that synchronized events are not split between the two components. □

4 Implementation and Experimental Results

To evaluate our approach we implemented Angluin's learning algorithm and the framework of [8] for the CSP model checker FDR2 (Failure Divergence Refinement) [18] to automatically verify a specification against a property based on the assume-guarantee proof rule [29]. Using the CSP semantics of CSP-OZ developed in [12], the specification is translated into the input language of FDR2.

The property we are aiming at can be described as follows: if at least one page votes with *NO*, all pages will *undo* their transaction. The CSP specification for *PROP* has already been given in Section 2. This property can now be checked for trace refinement against the full specification (with events not occurring in *PROP* hidden), i.e. using FDR2 syntax we check

```
assert PROP [T= SYSTEM \ {|request, execute, inform,
                           decide, result, acknowledge|}
```

We ran FDR2 on a Linux PC (Open SUSE 10.2) equipped with a 3 GHz Pentium 4 processor and 1 GB RAM. In Table 1, we give an overview of the computation times and sizes of the generated state spaces for using FDR2 to check *PROP*

- directly calling FDR2 on *System* without using compositional reasoning,
- using L* and assume-guarantee reasoning based on the *given* decomposition into *Coord* and *Pages*,
- using L* and assume-guarantee reasoning based on *our* decomposition based on three different sequential cuts:
  1. $Vot_1 \| Vot_2$ for the cut $\{vote\}$,
  2. $Dec_1 \| Dec_2$ for the cut $\{decide, vote\}$,
  3. $Inf_1 \| Inf_2$ for the cut $\{inform\}$.

We started with one instance of the component *Page* and incrementally increased N – the value used is given in the third column. The fourth column displays the verification time in seconds; column 5 and 6 indicate the size of the computed state space for components 1 and 2, respectively. One asterisk symbolizes that the machine ran out of memory due to exceeding its swap limit after approximately 90 minutes whereas two asterisks denote that there was no computation result after more than four hours.

Apparently, if we use our decomposition and L*, we are able to verify the property for a higher N compared to verification of the original specification. Furthermore, we achieve much better runtime results. The best results are achieved for the decomposition based on the cut $\{decide, vote\}$. Since the cut $\{decide, vote\}$ outperforms the cut $\{vote\}$, we can deduce that the smallest cut may not always achieve the best results. In particular, in our example, $\{vote\}$ leads to $V_C = \{votes\}$ – this variable is not represented in the corresponding set for $\{decide, vote\}$ since we are decomposing the specification at a point after which *votes* will never be used again.

Despite this there is no significant difference between the verification times for the different cuts we were looking at. The generated assumptions for N equals 2 for two of the cuts are depicted in Figure 3 (omitted from address parameters;

Table 1. Experimental Results for FDR2

System	L*	N	Time/sec	States Comp.1	States Comp.2
$Coord \parallel Pages$	no	1	<1	Full System: 47	
		2	<1	Full System: 1116	
		3	<1	Full System: 26190	
		4	6	Full System: 623376	
		5	227	Full System: 14984838	
		6	(*)	Full System: unknown	
$Coord \parallel Pages$	yes	1	1	9	65
		2	9	17	4353
		3	35	24	287496
		4	656	31	18974736
		5	(*)	37	(**)
$Vot_1 \parallel Vot_2$	yes	1	<1	17	12
		2	1	288	53
		3	3	4374	217
		4	5	64800	893
		5	21	949158	3673
		6	302	13845168	15053
		7	(*)	(**)	61417
$Dec_1 \parallel Dec_2$	yes	1	<1	23	11
		2	1	324	50
		3	2	4590	210
		4	5	66096	878
		5	19	956934	3642
		6	236	13891824	14990
		7	(*)	(**)	61290
$Inf_1 \parallel Inf_2$	yes	1	<1	29	8
		2	1	432	77
		3	2	6102	639
		4	4	85536	5201
		5	19	1197990	41799
		6	275	16831152	333137
		7	(*)	(**)	2640759

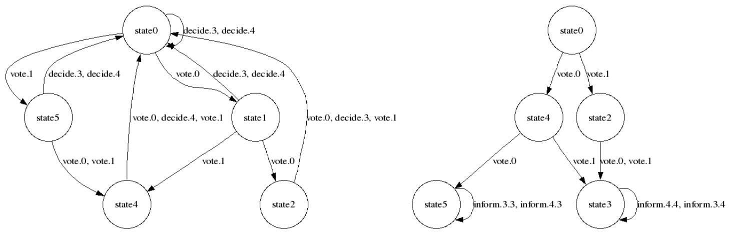

Fig. 3. Final assumptions for *PROP* based on Cuts $\{decide, vote\}$ and $\{inform\}$

0, 1 and 3, 4 are abstractions of *YES*, *NO* and *COMMIT*, *ABORT*, respectively). Runtime behaviour is worst for the given decomposition. Here, the results suffer from a larger generated assumption with more states and transitions.

5 Conclusion and Related Work

This paper presented an approach to compositional verification illustrated on specifications written in the integrated formal method CSP-OZ. We decomposed a specification such that the resulting components can efficiently be used for assume-guarantee reasoning. The decomposition is performed on the specification's dependence graph. We showed correctness of the approach. In the context of automatically generating assumptions we illustrated the technique on an example specification. Verification results are carried out using the CSP model checker FDR2.

Related work. Assume-guarantee reasoning was first introduced by Chandy and Misra [19] and Jones [17, 16]. Compositional verification for integrated formal methods undergoes intensive research. For CSP||B, a coupling of the B method with CSP, Treharne and Schneider explored compositional proof techniques [24] by also using FDR2 for verification of CSP processes.

The static analysis of a specification is the foundation for the decomposition technique proposed in this paper. Brueckner [4] defined a CSP-OZ dependence graph which we incorporated here. He used it to compute a specification slice [26] with respect to a certain property. Our decomposition technique is more closely related to the technique of program *chopping* [22] – we do not compute full specification slices but rather chop the specification up to the point at which the assumption holds.

Cobleigh et al. first used the L* algorithm in the context of automatic learning an assumption for compositional reasoning [8] in the domain of Labelled Transition Systems. We implemented their framework in our context for FDR2.

Alur and Nam [20] do assume-guarantee based reasoning in the context of symbolic model checking. They also use L* to automatically generate assumptions and decompose a given system. The decomposition is computed fully automatically in terms of hypergraph partitioning. In their semantic domain of *symbolic transition modules* based on solely boolean variables, they do not deal with control flow, synchronisation and dependence graphs.

Future work. This paper is intended to provide the basic concept for a decomposition and automated verification technique for CSP-OZ. There are many follow-up steps to be taken, some of which are described below.

The requirement that the dependence graph is sequential is quite strong. To relax this restriction, we need to deal with circularity within the DG. The technique proposed in this paper will thus be extended to specifications with a recursive structure. In this case, we aim to reuse the decomposition technique by defining *two* cuts determining the switch from the first to the second phase and vice versa. This will lead to the application of a symmetric assume-guarantee proof rule where two assumptions can be learned simultaneously [2].

Even though most of the steps within this framework can be performed automatically, such as the computation of the DG, the translation of a CSP-OZ class to the input language of FDR2 and the assumption learning, the definition of a cut is currently done by hand. To find an optimal cut in the sense of evenly distributed components we might use techniques presented in [14, 27] to compare several possible decompositions in terms of a *lattice of decomposition slices*. Other heuristics need to be defined and evaluated.

Finally, we aim at evaluating the approach on a much bigger case study.

Acknowledgement. We thank Ramsay Taylor for fruitful discussions on the topic and for correcting our English.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
2. Barringer, H., Giannakopoulou, D., Pasareanu, C.S.: Proof rules for automated compositional verification through learning. In: International Workshop on Specification and Verification of Component Based Systems, Finland (2003)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison (1987)
4. Brückner, I.: Slicing Integrated Formal Specifications for Verification. PhD thesis, Universität Paderborn (2008)
5. Brückner, I., Wehrheim, H.: Slicing an integrated formal method for verification. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 360–374. Springer, Heidelberg (2005)
6. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
7. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 97–108. ACM Press, New York (2006)
8. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
9. de Roever, W.P., Hanneman, U., Hooiman, J., Lakhneche, Y., Poel, M., Zwiers, J., de Boer, F.: Concurrency Verification. Cambridge University Press, Cambridge (2001)
10. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2(3), 155–173 (1982)
11. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: Formal Methods for Open Object-Based Distributed Systems (FMOODS 1997), vol. 2, pp. 423–438. Chapman and Hall, Boca Raton (1997)
12. Fischer, C., Wehrheim, H.: Model-checking CSP-OZ specifications with FDR. In: IFM, pp. 315–334 (1999)
13. Francez, N., Pnueli, A.: A proof method for cyclic programs. Acta Informatica 9(2) (1978)
14. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17(8), 751–761 (1991)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
16. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
17. Jones, C.B.: Tentative steps towards a development method for interfering programs. Transactions on Programming Languages and Systems 5(4), 596–619 (1983)

18. Formal Systems (Europe) Ltd. Failure divergence refinement: Fdr2 user manual (1997)
19. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. 7(4), 417–426 (1981)
20. Nam, W., Alur, R.: Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 170–185. Springer, Heidelberg (2006)
21. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
22. Reps, T.W., Rosay, G.: Precise interprocedural chopping. In: SIGSOFT FSE, pp. 41–52 (1995)
23. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
24. Schneider, S., Treharne, H.: Verifying controlled components. In: IFM, pp. 87–107 (2004)
25. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, Dordrecht (2000)
26. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
27. Tonella, P.: Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. Software Eng. 29(6), 495–509 (2003)
28. Weiser, M.: Programmers use slices when debugging. Commun. ACM 25(7), 446–452 (1982)
29. Wonisch, D.: Automatisiertes kompositionelles Model Checking von CSP Spezifikationen. Bachelor's thesis, Universität Paderborn (April 2008)