Distributed Synthesis

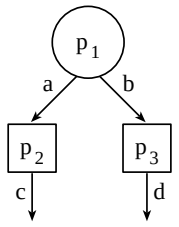
Jonathan Türpe

1 Introduction

In the synthesis problem one decides if a given temporal specification φ over a set of boolean formulas has an implementation, i.e., a finite state program, that satisfies φ. One can focus on several types considering this problem. The easiest type is considered the *closed synthesis*, i.e., a single-process wich does not interact with the environment. In contrast, *open synthesis* deals with the possible interaction with the environment. A third type is called *open synthesis with incomplete information* and has to handle interactions regarding the environment which remains possibly unseen by the process. This third most general type was solved for specifications given in CTL* by Kupferman and Vardi in 1997 [Kup.Var97].

The distributed synthesis generalizes the three types from the ones mentioned above. We consider not only one but several processes which can interact among each other in a certain way. This arrangement of processes and the structure of the interactions in-between them is called architecture. The *distributed synthesis problem* is given by a temporal specification φ over a set of boolean variables and an architecture A. The task is to decide whether there exists implementations of the single processes in A such that the overall behaviour satisfies φ or not.

That the distributed synthesis problem is indeed a challenging problem can be seen by the negative result given by Pnueli and Rosner in 1990 [Pnu.Ros90]. For the simple architecture given below and specifications in LTL the problem is undecidable.



In the following we will see that it is possible to part the classes of architectures into a class $\mathcal{D}$ which is decidable, and a class $\mathcal{U}$ which remains undecidable. The ideas and results are taken from [Fin.Sch05].

2 Distributed synthesis

To really understand and discuss the distributed synthesis problem the following definitions are needed.

Definition. *An* architecture A=(P, W, p_{env}, E, O, H) *is a tuple consisting of a set* P *of processes, a subset* $W \subset P$ *of white-box processes and an distinguished environment process* $p_{env} \in P \setminus W$. (P, E) *is a directed graph,* $O = \{O_e \mid e \in E\}$ *a set of non empty sets of output variables and* $H = \{H_p \mid p \in P\}$ *a set of hidden variables for each process. We assume that the hidden variables are pairwise disjoint from each other as well as from the sets in* O*. The same variable may occur on two separate edges to indicate broadcasting, but only if the edges are originated in the same note. For convenience we do not consider black-box processes without output variables, as they can be implemented trivially by mapping every input history to* $\emptyset$*. The set* $B := P \setminus W$ *contains the black-box processes and the environment,* $B^- := B \setminus p_{env}$ *is the set of all black-box processes but the environment.*

Definition. *An* implementation of a process p *is a function* $s_p : (2^{I_p})^* \to 2^{O_p}$ *called* strategy*, which maps every possible input history of the process to a possible output.*

Definition. *An* Implementation of an architecture A *is a set of strategies* $S = \{s_p \mid p \in B^-\}$*, for all black-box processes except the environment.*

Definition. *A* composition of a set of strategies *is a function* $s_Q : (2^{I_Q})^* \to 2^{O_Q}$*, where* I_Q *and* O_Q *denotes the common input or output of the processes respectively. The mapping is given according to the single strategies of the processes.*

In order to describe the meaning if an implementation satisfies a specification φ it is necessary to consider the computation tree. The directions of this tree are all possible combinations of outputs of the environment. Where the implementation maps to is fixed in the labels. Because the programs cannot recognize hidden variables of the environment, the definition needs the widening function *wide*. With the *xray* function the direction is written into the label.

Definition. *A* computation tree of an implementation S *is a* $2^{O \cup H}$ *labeled* $2^{O_{env}}$*-tree and given by:*
$< 2^{O_{env}}{}^*, \ell >= xray_{2^{O_{env}}}(wide_{2^{H_{env}}}(< (2^{O_{env} \setminus H_{env}})^*, s_{B^-} >))$

Definition. *A tuple* (A, φ) *consisting of an architecture* A *and a specification* φ *is* realizable *if there exists an implementation of* A *s.t. its computation tree satisfies.* φ

Definition. *An architecture is called* decidable *if there exists an algorithm which decides for all specifications* φ *if* (A, φ) *is realizable.*

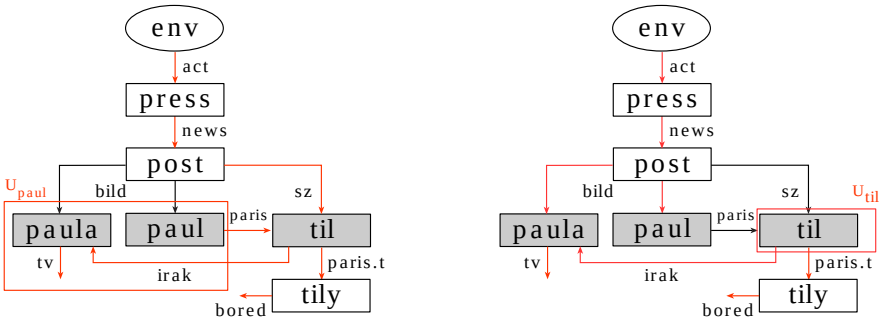
3 Partition of the classes of architectures

The partition of the classes of architectures is done by using the notion of *informedness of a process*. In case it is possible to compare each pair of black-box processes in a given architecture A among each other whether one has more or less information than the other, it turns out that A is decidable. On the other hand, if there is a pair of black-box processes in an architecture A which can not be compared in this sense, then A is an undecidable architecture.

Definition. *The preorder $p \preccurlyeq p'$ (Reading: p has more or equal information than p') is given by the following construction:*
Let $E_p = \{e \in E \mid O_e \not\subseteq I_p\}$ be the set of edges with information invisible to p and $U_p = \{q \in B \mid \neg\exists$ directed path from p_{env} to q in $(P, E_p)\}$ the processes which are not reachable by such edges then:
$p \preccurlyeq p' \leftrightarrow p' \in U_p$ for $p, p' \in B$

Example:

paul $\not\preccurlyeq$ til *til $\not\preccurlyeq$ paul*



This architecture is undecidable unless we modify it, by for example giving paul and paula the "sz" in addition.

Definition. *An architecture A is called (strictly)* ordered *if there exists a (bijective) surjective function, $f : B \to \mathbb{N}_n$ with $n \in \mathbb{N}$, s.t. $f^{-1}(1) = \{p_{env}\}$ and for all $p, p' \in B : f(p) \leq f(p')$ iff $p \preccurlyeq p'$.*

In the following two sections we will see that an architecture A is in the decidable class $\mathcal{D}$ of architectures if A can be ordered or in the undecidable class $\mathcal{U}$ otherwise.

4 The decidable class of architectures $\mathcal{D}$

To proof decidability of an ordered architecture, we transform at first any given architecture into a strictly ordered acyclic architecture without white-

box processes. After that we check by using an automata-based algorithm, whether the simplified architecture is decidable or not.

4.1 Architecture transformation

The transformation of the architecture is done in three steps. For each of the steps it holds that: (A_i, φ_i) is realizable iff (A_{i+1}, φ_{i+1}) is realizable. (A_0, φ_0) denotes the given problem (A, φ). The tuples (A_i, φ_i) with $i \in \{1, 2, 3\}$ denote the transformed ones according to the steps of the algorithm.

The three steps:

1. Clustering equally informed processes
2. Elimination of white-box processes
3. Elimination of feedback edges (i.e. an edge from a process to a better informed one)

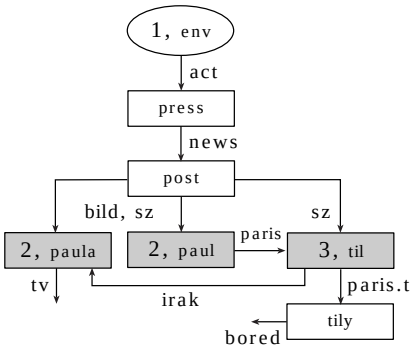
The algorithm and ideas of the proof:
Step 1
"$0 \leftrightarrow 1$" Assume that we have a strategy for A_0 that satisfies φ_0. For a cluster we take the composition of the implementations for the processes that are clustered together. This satisfies the specification $\varphi_1 := \varphi_0$. The other way around, if we have a strategy for A_1 which satisfies $\varphi_1 := \varphi_0$, we can copy the strategy of the cluster for each of the processes in it and restrict it then to the possible output variables. This leads to a strategy for A_0 which satisfies φ_0.

Step 2
"$1 \leftrightarrow 2$" Assume that we have an implementation of A_1 which satisfies φ_1. We take the best informed black-box process p' which provides the white-box process with information, and add edges from it so that all black-boxes still have the same input. As the white-box is already implemented, the strategy for the obtained process p'_{new} is just the composition of the old strategy and the "white" one. Formally this is done by turning the strategie for the white-box process s_w into an equivalent specification φ_w and add them to the original specification. In the other way round we just restrict the implementation to the possible outputs.

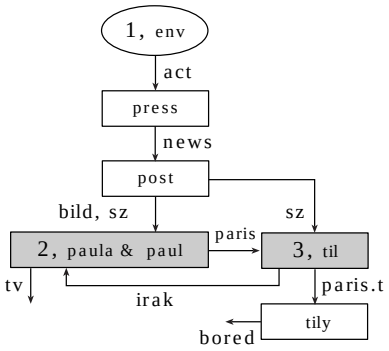
Step 3
"$2 \leftrightarrow 3$" In an analog way, a better informed process can always simulate a feedback. Vice versa having an implementation for A_3 that satisfies $\varphi_3 := \varphi_2$, the same implementation that ignores possible feedback edges is a perfect one for A_2.

Example:

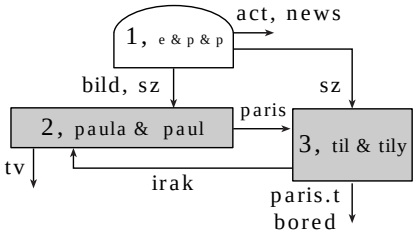
(A_0)



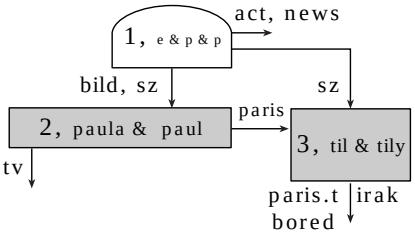
(A_1)



(A_2)



(A_3)



4.2 Automaton construction

Theorem. *Given a tuple (A, φ) where A is a strictly ordered architecture without white-boxes and φ a specification in CTL*, then:*

We can construct an alternating automaton $\mathcal{A}$ that recognizes a tree iff it is a computation tree of an implementation of A that satisfies φ.

Proof. We illustrate the idea of the proof using our example.
The first step is to translate the specification φ into an alternating automaton that accepts a $2^{\{a,n,b,s,t,p,p',i,b\}}$-labeled $2^{\{a,n,b,s\}}$-tree iff this tree satisfies φ. That way the label could be different from the direction of the environment and therefore trees are also accepted which are definitely not computation trees of A. In the second step we take care of this problem. The third step is done due to the possible hidden information of the environment. But nevertheless a trees accepted by this alternating automaton can have too many directions. The third process in this example does not have all the information *bild* and *sz*. The ensures that the output of a process depends only on its input we need further steps. The fourth step translates the alternating automaton to an equivalent nondeterministic automaton which

is necessary to perform the next step, but gives also an exponential blow-up in the number of states. The fifth step is to construct an alternating automaton that accepts a $2^{\{p',i,b\}}$-labeled $2^{\{b,s\}}$-tree iff there exists a $2^{\{t,p,p',i,b\}}$-labeled $2^{\{b,s\}}$-tree which is accepted by the former automaton. Now we use the third step again to gain an automaton that accepts $2^{\{p',i,b\}}$-labeled $2^{\{p,s\}}$-trees iff this is a computation tree of the third process and the widening of the tree is accepted by the former automaton. In general we have to repeat step 4, 5 and 3 until the process with the least information is reached and we are able to take care about its incomplete information. If the resulted automaton is not empty, then (A, φ) is satisfiable. Going the steps back again we can construct an implementation for A which satisfies φ.

1. $\varphi \rightsquigarrow \mathcal{A}_1$ over $2^{\{a,n,b,s,t,p,p',i,b\}}$-labeled $2^{\{a,n,b,s\}}$-trees
2. $\mathcal{A}_1 \rightsquigarrow \mathcal{A}_2$ over $2^{\{t,p,p',i,b\}}$-labeled $2^{\{a,n,b,s\}}$-trees
3. $\mathcal{A}_2 \rightsquigarrow \mathcal{A}_3$ over $2^{\{t,p,p',i,b\}}$-labeled $2^{\{b,s\}}$-trees
4. $\mathcal{A}_3 \rightsquigarrow \mathcal{N}_3$ nondeterministic automaton
5. $\mathcal{N}_3 \rightsquigarrow \mathcal{A}_4$ over $2^{\{p',i,b\}}$-labeled $2^{\{b,s\}}$-trees
6. $\mathcal{A}_4 \rightsquigarrow \mathcal{A}'_3$ over $2^{\{p',i,b\}}$-labeled $2^{\{p,s\}}$-trees

□

5 The undecidable class of architectures $\mathcal{U}$

Theorem. *Let A be an architecture, s.t. A can not be ordered and the language for the specifications is either in CTL or in LTL.*
The architecture A is undecidable.

Idea of the proof: Using a reduction from the halting problem.

6 Conclusion

We showed that the distributed synthesis problem can be solved for specifications in CTL^* if and only if the architecture A is in a special class $\mathcal{D}$ of architectures. Furthermore we showed how to check whether A is in this class or not.

References

[Fin.Sch05] Finkbeiner, B., Schewe, S.: Uniform distributed synthesis. In: Proc. LICS, IEEE Computer Society Press (2005) 321- 330

[Kup.Var97] O. Kupferman and M. Y. Vardi. Synthesis with incomplete information. In Proc. ICTL'97, 1997.

[Pnu.Ros90] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In Proc. FOCS'90, pages 746-757, 1990.