An Algorithmic Framework for Synthesis of Concurrent Programs

E. Allen Emerson and Roopsha Samanta

Dept. of Computer Science and Computer Engineering Research Centre,
University of Texas, Austin, TX 78712, USA.
{emerson,roopsha}@cs.utexas.edu

Abstract. We present a framework that takes unsynchronized sequential processes along with a temporal specification of their global concurrent behaviour, and automatically generates a concurrent program with synchronization code ensuring correct global behaviour. The synthesized synchronization code is based on monitors with `wait` and `notify` operations on condition variables, and mutual-exclusion locks. Novel aspects of our framework include realistic low-level synchronization implementations, synthesis of both simple coarse-grained synchronization and more complex fine-grained synchronization, and accommodation of both safety and liveness in global correctness properties. The method is fully automatic as well as sound and complete.

1 Introduction

We postulate design and employment of automated synthesis engines for the most precarious component of a concurrent program - the synchronization code. Given unsynchronized *skeletons* of sequential processes $P_1, \ldots, P_n$, and a temporal specification ϕ of their global concurrent behaviour, our framework *automatically* generates synchronized skeletons, $\overline{P}_1, \ldots, \overline{P}_n$, such that the resulting concurrent program $\overline{P}_1 \mathbin{/\!/} \ldots \mathbin{/\!/} \overline{P}_n$ is guaranteed to exhibit the desired behaviour. This is effected in two steps. The first step involves computer-aided construction of a model $\mathcal{M}$ for the specified behaviour of the concurrent program based on $P_1, \ldots, P_n$, and extraction of *synchronization skeletons* $P_1^s, \ldots, P_n^s$, with high-level synchronization actions (guarded commands), such that $P_1^s \mathbin{/\!/} \ldots \mathbin{/\!/} P_n^s \models \phi$. The second step comprises a correctness-preserving mechanical compilation of the high-level synchronization actions into synchronization code based on lower-level primitives such as monitors and mutual-exclusion (mutex) locks.

The first step in our framework could be completed by manually constructing a high-level solution, and then verifying its correctness using a model checker (cf. [15]). However, the lack of automation in constructing

the high-level solution is a potentially serious drawback as it may necessitate multiple iterations of manual (re-)design, verification, and manual debugging and correction. We propose a substantial improvement to this approach that results in a fully algorithmic framework. By specifying the system temporally, we can apply the method of [7, 6] to algorithmically synthesize the high-level solution guaranteed to meet the temporal specification. We alleviate the user's burden of specification-writing by automatically inferring local temporal constraints, describing process behaviour, from a state-machine based representation of the unsynchronized processes.

We provide the ability to synthesize coarse-grained synchronization code with a single monitor (and no mutex locks), or fine-grained synchronization code with multiple monitors and mutex locks. It is up to the user to choose an appropriate granularity of atomicity that suitably balances the trade-off between concurrency and overhead for a particular application/system architecture. This is an important feature of our framework as programmers often restrict themselves to using coarse-grained synchronization for its inherent simplicity. In fact, manual implementations of synchronization code using `wait`/`notify` operations on condition variables are particularly hard to get right in the presence of multiple locks. We establish the correctness of both translations - guarded commands to coarse-grained synchronization and guarded commands to fine-grained synchronization - with respect to typical concurrency properties that include both safety properties (e.g., mutual exclusion) and liveness properties (e.g., starvation-freedom).

We further establish soundness and completeness of the overall proposed methodology. Thus, our generated concurrent programs are *correct-by-construction*, with no further verification effort required. Moreover, if the specification as a whole is consistent, a correct concurrent program will be generated. We have developed a tool for the compilation of synchronization skeletons into concurrent Java programs with both coarse-grained and fine-grained synchronization. We used the tool successfully to synthesize synchronization code for an airport ground traffic simulator program, and some well-known synchronization problems such as readers-writers and dining philosophers. We emphasize that the synchronization code generated by our framework can be translated into programs written using PThreads or in C# as well.

The most important contribution of our work is the combination of an algorithmic front end for synthesizing a high-level synchronization solution, with an algorithmic back end that yields a readily-implementable

low-level synchronization solution. We use the CTL-based decision procedure from [7, 6] because it is handy and available. But an algorithmic front end could be supplied in many alternative ways; for instance, any linear temporal logic (LTL) decision procedure could be used. Other novel ingredients of our fully algorithmic framework include provably correct translations of high-level to low-level synchronization, synthesis of both coarse-grained and fine-grained solutions, and accommodation of both safety and liveness in global correctness properties. Moreover, our method is sound and complete.

The paper is structured as follows. We explain our algorithmic framework using an example concurrent program in Sec. 2. We discuss extensions and experimental results in Sec. 3 and conclude with a review of related work in Sec. 4.

2 Algorithmic Framework

In this section, we present an overview of our approach for concurrent programs based on two processes, using a single-reader-single-writer (RW) example. We refer the reader to [8] for a more detailed treatment of our formal framework and algorithms.

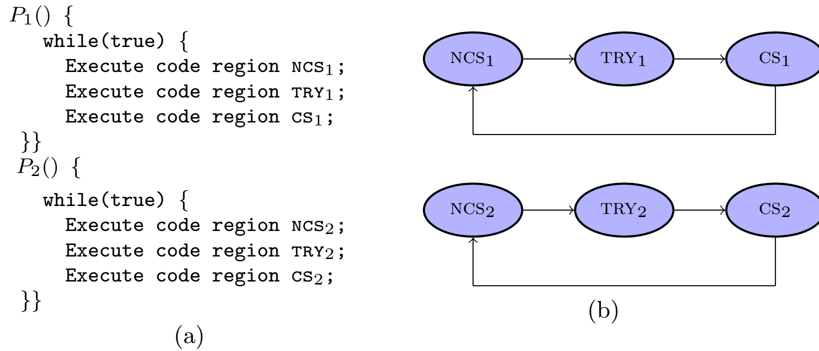


Fig. 1: Synchronization-free skeletons of two processes: reader P_1 and writer P_2

Table 1: Specification of synchronization for single-reader-single-writer problem

Mutual exclusion: $\mathsf{AG}(\neg(\textsc{cs}_1 \wedge \textsc{cs}_2))$.
Absence of starvation for reader P_1, provided writer P_2 remains in its non-critical region: $\mathsf{AG}(\textsc{try}_1 \Rightarrow \mathsf{AF}(\textsc{cs}_1 \vee \neg\textsc{ncs}_2))$.
Absence of starvation for writer: $\mathsf{AG}(\textsc{try}_2 \Rightarrow \mathsf{AF}\,\textsc{cs}_2)$.
Priority of writer over reader for outstanding requests to enter the critical region: $\mathsf{AG}((\textsc{try}_1 \wedge \textsc{try}_2) \Rightarrow \mathsf{A}[\textsc{try}_1\, \mathsf{U}\, \textsc{cs}_2])$.

We assume that we are given the synchronization-free skeletons of sequential processes P_1 and P_2 and a temporal specification ϕ of their de-

sired global behaviour. The synchronization-free skeletons of the reader process P_1 and the writer process P_2 are as shown in Fig. 1a. Both processes have three *code regions* - 'non-critical' (NCS), 'trying' (TRY) and 'critical' (CS); the control-flow between these code regions can be encoded as state-machines, as shown in Fig. 1b. Each code region may represent a *terminating* sequential block of code, which is irrelevant for the synthesis of synchronization, and hence suppressed within a single state. The set of properties that the concurrent program composed of P_1 and P_2 must guarantee are shown in Table 1. It is easy to see that in the absence of synchronization $P_1 /\!\!/ P_2 \not\models \phi$, where ϕ represents the conjunction of the properties in Table 1. Our goal is to modify P_1 and P_2 by inserting synchronization code, to obtain $\overline{P}_1$ and $\overline{P}_2$, such that $\overline{P}_1 /\!\!/ \overline{P}_2 \models \phi$.

We propose an automated framework to do this in two steps. The first step entails computer-aided construction of a high-level solution with synchronization actions based on guarded commands. The second step comprises a correctness-preserving, mechanical translation into a low-level solution based on monitors (along with `wait` and `notify` operations on condition variables) and mutex locks.

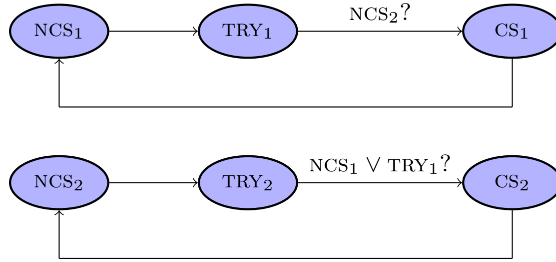


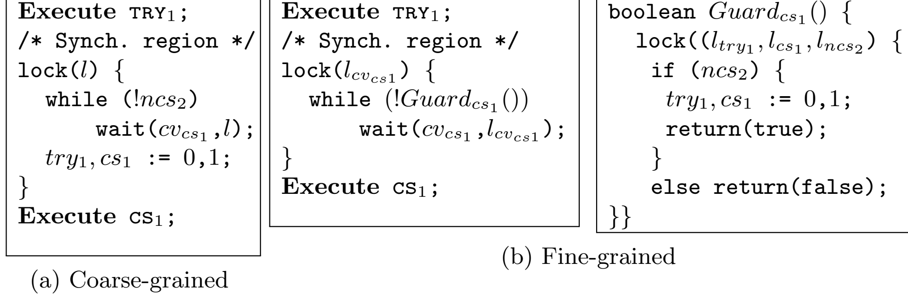
Fig. 2: Synchronization skeletons P_1^s and P_2^s for reader P_1 and writer P_2

For the first step, we mechanically translate the state-machine representations of P_1 and P_2 into equivalent CTL formulae. We then use the methodology presented in [7] to: (1) synthesize a global model $\mathcal{M}$ for the specified behaviour of the concurrent program based on P_1 and P_2, such that $\mathcal{M} \models \phi$, and (2) derive the synchronization skeletons, P_1^s and P_2^s (see Fig. 2) from $\mathcal{M}$. We refer the interested reader to [8, 7] for details about the synthesis of $\mathcal{M}$ and P_1^s, P_2^s. For our current purpose, it suffices to note that each transition between two sequential code regions in the synchronization skeleton of a process is labeled with a guarded command of the form $G? \rightarrow A$, consisting of an enabling condition G, evaluated atomically, and a corresponding set of actions A to be performed atomi-

cally if G evaluates to *true*. A guard is a predicate on the current state (code region) of all processes and the values of shared synchronization variables, $x_1, \ldots, x_m$ (this tuple is often denoted as $\bar{x}$), which may be introduced during the synthesis of $\mathcal{M}$. An action is a parallel assignment statement that updates the values of the $\bar{x}$ variables. All guards with the same action are merged into one transition label. An omitted guard is interpreted as *true* in general. In the RW example (Fig. 2), there are no actions as no $\bar{x}$ variables were introduced during the synthesis of $\mathcal{M}$.

In the second step of our approach, we mechanically compile the guarded commands of P_1^s and P_2^s into either coarse-grained or fine-grained synchronization code for P_1 and P_2, as desired. The resulting processes are denoted as P_1^c, P_2^c (coarse-grained) or P_1^f, P_2^f (fine-grained). In both cases, we introduce Boolean shared variables, ncs_1, try_2 etc., to represent the code regions NCS_1, TRY_2 etc., of each sequential process. We also introduce mutex locks and monitors along with conditions variables for synchronization. For the program $P_1^c \mathbin{/\!\!/} P_2^c$, which has a coarser level of lock granularity, we declare a single lock l for controlling access to shared variables and condition variables. For the program $P_1^f \mathbin{/\!\!/} P_2^f$ with a finer level of lock granularity, we allow more concurrency by declaring separate mutex locks l_{ncs_1}, l_{try_2} etc., for controlling access to each Boolean shared variable ncs_1, try_2 etc. (and each shared synchronization variable, when necessary). We further define separate monitor locks $l_{cv_{cs_1}}$, $l_{cv_{cs_2}}$ for the condition variables cv_{cs_1}, cv_{cs_2} to allow simultaneous processing of different condition variables.

The modifications to each process are restricted to insertion of *synchronization regions* between the sequential code regions of the process. We refer the reader to Fig. 3a for an example coarse-grained synchronization region (between code regions TRY_1 and CS_1 in P_1). Note that we find it convenient to express locks, as `lock(l){...}` (in a manner similar to Java's `synchronized` keyword), wherein l is a lock variable, '{' denotes *lock acquisition* and '}' denotes *lock release*. The implementation of a coarse-grained synchronization region for the RW example involves acquiring the monitor lock l and checking, within the monitor, if the guard G (ncs_2 in Fig. 3a) for entering the next code region is enabled. While the guard is $false$, P_1^c waits for P_2^c to be in an enabling code region. This is implemented by associating a condition variable cv (cv_{cs_1} in Fig. 3a) with the guard for the next code region. Thus while G is $false$, P_1^c *waits* till P_2^c *notifies* it that G could be *true*. If the guard G is *true*, P_1^c updates the values of (the $\bar{x}$ variables, when present, and) the shared Boolean variables in parallel to indicate that it is effectively in the next



(a) Coarse-grained

(b) Fine-grained

Fig. 3: Coarse and fine-grained synchronization regions between code regions TRY_1 and CS_1 of reader process P_1

code region and releases the monitor lock. Before the lock release, P_1^c, in general, sends a notification signal corresponding to every guard (i.e. condition variable) of P_2^c which may be changed to *true* by P_1^c's shared variables update - there is no such notification in Fig. 3a as the update does not change any guard of P_2^c to *true*. If the guard for a code region is always *true*, e.g., code region TRY_1, then we do not need to check its guard, and hence, do not need a condition variable corresponding to the guard of the code region.

While fine-grained locking can typically be achieved by careful definition and nesting of multiple locks, one needs to be especially cautious in the presence of monitor locks for various reasons. For instance, upon execution of `wait(`cv,l`)` in a nested locking scheme, a process only releases the lock l before going to sleep, while still holding all outer locks. This can potentially lead to a deadlock. A fine-grained synchronization region synthesized in our approach (see Fig. 3b for an example fine-grained synchronization region preceding code region CS_1 in P_1), circumvents these issues by utilizing a separate subroutine to evaluate the guard G. In this subroutine, P_1^f first acquires all *necessary* mutex locks, corresponding to all shared variables accessed in the subroutine. These locks are acquired in a strictly nested fashion in a predecided fixed order to prevent deadlocks. We use `lock(`$l_1, l_2, \ldots$`)`$\{\ldots\}$ to denote the nested locks `lock(`l_1`)`$\{$ `lock(`l_2`)`$\{\ \ldots\}\}$, with l_1 being the outermost lock variable. The subroutine then evaluates G and returns its value to the main body of P_1^f. If found *true*, the subroutine also performs an appropriate parallel update to the shared variables similar to the coarse-grained case. The synchronization region in the main body of P_1^f acquires the relevant monitor lock ($l_{cv_{cs_1}}$ in Fig. 3b) and calls its guard-computing subroutine within a `while`

loop till it returns $true$, after which it releases the monitor lock. If the subroutine returns $false$, the process waits on the associated condition variable (cv_{cs_1} in Fig. 3b). Each notification signal for a condition variable, on which the other process may be waiting, is sent out by acquiring the corresponding monitor lock.

2.1 Correctness of Synthesis

Let $\mathcal{M}^c$ and $\mathcal{M}^f$, be the global models corresponding to $P_1^c \mathbin{/\!/} P_2^c$ and $P_1^f \mathbin{/\!/} P_2^f$, respectively. We have the following *Correspondence Lemmas*:

Lemma 1. *[Coarse-grained Correspondence] Given an* $ACTL \setminus X$ *formula* ϕ, $\mathcal{M} \models \phi \Rightarrow \mathcal{M}^c \models \phi$.

Lemma 2. *[Fine-grained Correspondence] Given an* $ACTL \setminus X$ *formula* ϕ, $\mathcal{M} \models \phi \Rightarrow \mathcal{M}^f \models \phi$.

The proofs are based on establishing *stuttering simulations* between the models (cf. [8][1]). Note that the models are not stuttering bisimilar, and hence our compilations do not preserve arbitrary $CTL \setminus X$ properties. This is not a problem, as most global concurrency properties of interest (see Table 1) are expressible in $ACTL \setminus X$.

Theorem 1. *[Soundness]: Given unsynchronized skeletons* P_1, P_2, *and an* $ACTL \setminus X$ *formula* ϕ, *if our method generates* P_1^c, P_2^c *(resp.,* P_1^f, P_2^f*), then* $P_1^c \mathbin{/\!/} P_2^c \models \phi$ *(resp.,* $P_1^f \mathbin{/\!/} P_2^f \models \phi$*).*

Theorem 2. *[Completeness]: Given unsynchronized skeletons* P_1, P_2, *and an* $ACTL \setminus X$ *formula* ϕ, *if the temporal specifications describing* P_1, P_2 *and their global behaviour* ϕ *are consistent as a whole, then our method constructs* P_1^c, P_2^c *(resp.,* P_1^f, P_2^f*) such that* $P_1^c \mathbin{/\!/} P_2^c \models \phi$*(resp.,* $P_1^f \mathbin{/\!/} P_2^f \models \phi$*).*

The soundness follows directly from the soundness of the synthesis of synchronization skeletons [7, 6], and from the above Correspondence Lemmas. The completeness follows from the completeness of the synthesis of synchronization skeletons for overall consistent specifications and from the completeness of the compilation of guarded commands to coarse-grained and fine-grained synchronization.

[1] While we choose to restrict our attention to the preservation of $ACTL \setminus X$ formulas here, we can show that the translations from $\mathcal{M}$ to $\mathcal{M}^c$ and $\mathcal{M}^f$ actually preserve all $ACTL^* \setminus X$ properties, as well as CTL^* properties of the form Ah or Eh, where h is an $LTL \setminus X$ formula.

3 Extensions and Experiments

The synthesis of synchronization skeletons in the first step in our framework can be extended directly to handle an arbitrary number n of sequential processes. While the direct extension based on [7] can be exponential in the length of ϕ and in n, the decision procedure in [6], corresponding to the subset of CTL used in this paper, is polynomial in the length of ϕ. Moreover, we can use the approaches of [2, 1] which avoid building the entire global model (exponential in n), and instead compose interacting process pairs to synthesize the synchronization skeletons. The compilation of guarded commands into coarse-grained and fine-grained synchronization code can be extended in a straight-forward manner to $n > 2$ processes. We emphasize that this compilation acts on individual skeletons directly, without construction or manipulation of the global model, and hence circumvents the state-explosion problem for arbitrary n.

We have implemented a prototype synthesis tool [8] in Perl, which automatically compiles synchronization skeletons into concurrent Java programs based on both coarse-grained and fine-grained synchronization. We used the tool successfully to synthesize synchronization code for an example airport ground traffic simulator (AGTS) program (cf. [8]), and for several configurations of n-process mutual exclusion, readers-writers, dining philosophers, etc..

Table 2: Experimental Results

Program	Granularity	Norm. Run. Time
2-plane AGTS	Coarse	1
	Fine	0.92
1-Reader, 1-Writer	Coarse	1
	Fine	0.79
2-process Mutex	Coarse	1
	Fine	1.08
2-Readers, 3-Writers	Coarse	1
	Fine	1.14

Our experiments were run on a quad-core 3.4GHz machine with 4GB of RAM. The time taken by the tool to generate these small examples was a few milliseconds. We present the normalized running times of some of the generated examples in Table 2. As expected, the fine-grained synchronization version does not always outperform the coarse-grained synchronization version. In particular, it suffers in the 2-Readers, 3-Writers example due to excessive locking overhead.

4 Concluding Remarks

Our framework for concurrent program synthesis: (a) caters for both safety and liveness, (b) is fully algorithmic, (c) constructs a high-level

synchronization solution, (c) yields a low level solution based on widely used synchronization primitives, (d) can generate both coarse-grained and fine-grained low-level solutions, and (e) is provably sound and complete.

Early work on synthesis of high-level concurrent programs from temporal specifications [7] utilized decision procedures but had little practical impact due to unrealistic synchronization primitives. Other work inferring high level synchronization using guarded commands [13] or atomic sections [14], is limited to safety specifications. Moreover, it can be shown that such synthesis methods that rely on pruning a global product graph [10, 13, 14] cannot work in general for liveness.

On the other end of the spectrum, the important papers [5, 15] describe a needed mapping of a high-level system into a low-level, coarse-grained system, akin to ours. But these frameworks are less flexible. They do not yield low-level fine-grained solutions; they do not treat liveness properties; and, because they are not fully algorithmic, they fail to ensure correctness-by-design. Instead, these papers are verification-driven, and involve verifying either the synthesized implementation [5] or the manually-written high-level implementation [15]. In contrast, our approach is the first to provably translate a high-level system into correct low-level systems for both coarse- and fine-grained solutions, thereby eliminating the need for verification. The low-level global models are guaranteed correct by our Correspondence Lemmas.

Among papers that do address refinement of locking granularity, are [3], which translates guarded commands, into synchronization based on atomic reads and atomic writes, and papers on compiler-based lock inference for atomic sections ([9], [4] etc.). Unlike in [3], our framework does not manipulate or generate the global model corresponding to either the coarse-grained or fine-grained solutions. The lock-inference papers [9], [4] rely on the availability of high-level synchronization in the form of atomic sections, and do not, in general, support monitors and condition variables. Sketching [12], a search-based program synthesis technique, is also a verification-driven approach, which can be used to synthesize optimized implementations of synchronization primitives, e.g. barriers, from partial program sketches.

We remark that these approaches and ours are oriented towards closed systems[2], which include classical synchronization problems and have been used to capture many real-world software systems.

[2] In contrast, another active thread of foundational research [11] investigates synthesis of open systems.

References

1. Attie, P.C.: Synthesis of Large Concurrent Programs via Pairwise Composition. In: Proceedings of Conference on Concurrency Theory (CONCUR). pp. 130–145. ACM (1999)
2. Attie, P.C., Emerson, E.A.: Synthesis of Concurrent Systems with Many Similar Sequential Processes. In: Proceedings of Principles of Programming Languages (POPL). pp. 191–201. ACM (1989)
3. Attie, P.C., Emerson, E.A.: Synthesis of Concurrent Systems for an Atomic Read/Atomic Write Model of Computation. In: Proceedings of Principles of Distributed Computing (PODC). pp. 111–120. ACM (1996)
4. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring Locks for Atomic Sections. In: Proceedings of Programming Language Design and Implementation (PLDI). pp. 304–315. ACM (2008)
5. Deng, X., Dwyer, M.B., Hatcliff, J., Mizuno, M.: Invariant-based Specification, Synthesis, and Verification of Synchronization in Concurrent Programs. In: Proceedings of International Conference on Software Engineering (ICSE). pp. 442–452. ACM (2001)
6. Emerson, E.A., Sadler, T., Srinivasan, J.: Efficient Temporal Reasoning. In: Proceedings of Principles of Programming Languages (POPL). pp. 166–178. ACM (1989)
7. Emerson, E.A., Clarke, E.M.: Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. Sci. Comput. Program. 2(3), 241–266 (1982)
8. Emerson, E.A., Samanta, R.: An Algorithmic Framework for Synthesis of Concurrent Programs. `http://www.cerc.utexas.edu/~roopsha/synthsync.html`
9. Emmi, M., Fishcher, J.S., Jhala, R., Majumdar, R.: Lock Allocation. In: Proceedings of Principles of Programming Languages (POPL). pp. 291–296 (2007)
10. Janjua, M.U., Mycroft, A.: Automatic Correction to Safety Violations in Programs. In: Proceedings of Thread Verification (2006)
11. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proceedings of Principles of Programming Languages (POPL). pp. 179–190. ACM (1989)
12. Solar-Lezama, A., Rabbah, R., Bodik, R., Ebcioglu, K.: Programming by Sketching for Bit-streaming Programs. In: Proceedings of Programming Language Design and Implementation (PLDI). pp. 281–294. ACM (2005)
13. Vechev, M.T., Yahav, E., Yorsh, G.: Inferring Synchronization under Limited Observability. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS). pp. 139–154 (2009)
14. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-Guided Synthesis Of Synchronization. In: Proceedings of Principles of Programming Languages (POPL). pp. 327–388 (2010)
15. Yavuz-Kahveci, T., Bultan, T.: Specification, Verification, and Synthesis of Concurrency Control Components. In: Proceedings of International Symposium on Software Testing and Analysis (ISSTA). pp. 169–179. ACM (2002)