Modular System Synthesis

Kanghee Park, Keith J.C. Johnson, Loris D’Antoni, Thomas Reps
University of Wisconsin–Madison
Madison, USA
{khpark, keithj, loris, reps}@cs.wisc.edu

Abstract—This paper describes a way to improve the scalability of program synthesis by exploiting modularity: larger programs are synthesized from smaller programs. The key issue is to make each “larger-created-from-smaller” synthesis sub-problem be of a similar nature, so that the kind of synthesis sub-problem that needs to be solved—and the size of each search space—has roughly the same character at each level. This work holds promise for creating program-synthesis tools that have far greater capabilities than currently available tools, and opens new avenues for synthesis research: how synthesis tools should support modular system design, and how synthesis applications can best exploit such capabilities.

I. INTRODUCTION

In program synthesis, the goal is to automatically (or semi-automatically) create programs that match high-level intents provided by a user—e.g., logical specifications or input-output examples. To date, however, synthesis tools cannot contend with large programs because they require synthesizing (or at least reasoning about) a program in its entirety.

The obvious direction is to try to exploit compositionality and synthesize larger programs by having them invoke other (already synthesized) programs. Consider for example the problem of writing a program for a ticket-vendor application that can, among other things, issue and reserve tickets. Building such a system requires creating modules for various data structures—perhaps a stack and queue—and using these modules in a top-level module that processes ticket requests. It is natural to ask whether such modules can be synthesized separately—i.e., in a compositional fashion.

The fundamental question is

A solution to this question is surprisingly tricky to envisage. Most existing synthesis approaches require having a concrete semantics or implementation in hand when reasoning about modules, components, APIs, etc. [5], [18], [20], and such synthesis tools end up reasoning about the entire program all the way down to its lowest-level components. Not only is this approach in fundamental opposition to the “similar-nature/similar-size” principle articulated above, it makes synthesis increasingly hard as more modules are considered.

Instead, when code is synthesized for some module $M$, all reasoning about lower-level modules $\{M_i\}$ on which $M$ directly depends should be carried out in a way that is agnostic about the implementations of $\{M_i\}$. This observation leads us to pose two related challenges: (i) How can one carry out program synthesis without having in-hand details about the implementations of lower-level modules? (ii) How can one ensure that each synthesis problem results in code that is independent of the implementations of lower-level modules?

In this paper, we present the case for the following thesis:

Program synthesis can scale using modular system design.

Modular system design is one of the most important concepts in designing software. A system should be organized in a layered fashion, where information hiding is used to hide implementation choices [16]. The information-hiding principle intuitively states that each module exports an interface that does not reveal specific implementation choices used inside the module, and changing the module’s implementation should not force any changes to be made to other modules.

Programmers practice modular system design, or at least aspire to it. In essence, our goal is to provide a level of automation for what good programmers do manually. Of course, we are not trying to automate everything. What is left in the hands of the programmer are architectural decisions and specifications of the intended behavior of individual modules. The programmer is responsible for the overall organization of the system’s design, and must decide such issues as: What are the layers in the system? What are the implementation choices in a given layer (such as choices about data structures and data representations)? What operations are exposed in each layer, and what is the intended behavior of each operation?

We identify two opportunities for providing automation for each module and, as a key contribution of this paper, we formally define these synthesis problems.

Module-Implementation Synthesis. Synthesis can be helpful in creating the implementations of the various functions in each module from some specifications. The key difference from traditional synthesis problems is that implementation details of “lower” modules are not available. Instead, one only has access to implementation-agnostic specifications of the semantics of such modules.

Module-Specification Synthesis. Because modules can only expose their semantics to other modules in a way that does not reveal their implementation details, it can be challenging
to come up with such semantic definitions. We propose to automate the creation of such implementation-agnostic semantic definitions using synthesis, namely, synthesis of formulas.

Note the role of the second kind of synthesis problem: its results provide part of the specification when one moves on to the task of synthesizing the implementation of functions in the next module. By analogy with the Paul Simon lyric “one man’s ceiling is another man’s floor” [19], we have “one module’s semantics is another module’s primitives.”

We call this approach modular system synthesis (MoSS). The visibility restrictions of information hiding provide the key for MoSS to achieve the objective of making synthesis scalable via “similar-nature/similar-size” sub-problems: both of our synthesis problems concern a single module of the system, and a single module’s implementation only. By concealing the implementation of lower-level modules, MoSS ensures that the formula representing the semantics of these layers remains independent of the size of the “accumulated” system as we move to higher-level layers. Moreover, MoSS retains the usual benefit of modular system design, namely, it results in software that (usually) can be readily adapted—in this context, re-synthesized—as requirements change.

This paper contributes both a framework and solidifying the concept of contract-based design in the context of program synthesis, which abstracts components or sub-systems based on their interfaces. Notably, the study of interface compatibility and composition has not been extensively explored in the context of program synthesis, opening up many opportunities for future developments. Specifically, using the aforementioned ticket-vending application as an example (§II), it (i) defines modular system synthesis (§III); (ii) defines the two kinds of synthesis problems that arise in MoSS (§IV); and (iii) describes a proof-of-concept system, called MoSSKit, that achieves these goals (§V).

MoSSKit is based on two existing program-synthesis techniques: JLIBSKETCH [14] a program-sketching tool that supports algebraic specifications, and SPIRO [15] a tool for synthesizing precise specifications from a given codebase. We used MoSSKit to carry out case studies based on two-layer modular synthesis problems from Mariano et al. [14], which demonstrated that concealing lower-level components can be advantageous in reducing the complexity of the synthesis problem. Expanding upon their work, our case study in §V-B further explored scenarios involving multiple layers. MoSS exhibits even better scalability compared to scenarios where executable semantics for all lower layers are exposed. A further case study based on Mariano et al. in §V-D also highlights the challenges of writing correct specifications. Our framework and the act of performing synthesis for both the implementations and specifications of the modules unveiled bugs in the modules synthesized by Mariano et al. and in the module’s specifications, which they manually wrote.

§VI discusses related work. §VII concludes.

II. ILLUSTRATIVE EXAMPLE

We present an experiment that illustrates the various aspects of MoSS. The problem to be solved is as follows: Synthesize a simple ticket-vendor application that supports the operations prepSales, resTicket, issueTicket, soldOut, numTicketsRem, and numWaiting. (To simplify matters, we assume it is not necessary to cancel a reservation.)

A. A Modular TicketVendor Implementation

We decompose the system into three modules (Fig. 1): Module 3: The TicketVendor module uses a Queue of reservations to implement the aforementioned operations. Module 2: The Queue module implements the operations emptyQ, enq, front, deq, sizeQ, and isEmptyQ. In our setting, a Queue is implemented using two stacks [12].

Module 1: The Stack module implements the operations emptyS, push, top, pop, sizeS, and isEmptyS. In our setting, a Stack is implemented using linked-list primitives of the programming language.

Moreover, the implementation of each module is to abide by the principle of information hiding: (i) The TicketVendor module can use operations exposed by Queue, but their actual implementations are hidden in Module 2. (ii) The Queue module can use operations exposed by Stack, but their actual implementations are hidden in Module 1.

B. The Input of Modular TicketVendor Synthesis

A MoSSKit user supplies the following information:

Architectural-design choices:
- The decomposition of the problem into TicketVendor, Queue, and Stack modules (gray boxes in Fig. 1).
- Which operations are to be exposed by each module, denoted by \( P[\text{module}] \)—e.g., in Fig. 1, the Queue module exposes \( P[\text{Queue}] \), which contains enq and deq operations, but not push and pop operations on the underlying stacks.

Data-structure/data-representation choices:
Module 3: TicketVendor uses a Queue.
Module 2: A Queue is implemented using two Stacks.
Module 1: A Stack is implemented using a linked list.

These choices are shown by the green boxes underneath each module in Fig. 1. For example, the Queue module is built on top of the Stack module. However, only the Stack interface—i.e., the function symbols in \( P[\text{Stack}] \) and its (potentially synthesized) implementation-agnostic specification \( \varphi^\text{sem}_{\text{Stack}} \)—is accessible by the Queue module.

Specifications of the module-specific synthesis problems:
Module 3: Specifications of the behaviors of prepSales, resTicket, issueTicket, soldOut, numTicketsRem, and numWaiting in terms of the exposed Queue operations (and possibly other TicketVendor operations). For example, the implementation-specific specifications for the

\[ \text{The invariant is that the second Stack holds a prefix of the Queue's front elements, with the top element of the second Stack being the Queue's front-most element. The first Stack holds the Queue's back elements—with the top element of the first Stack being the Queue's back-most element.} \]
**Fig. 1.** Organization of the modular TicketVendor synthesis problem: user-supplied inputs are shown in solid boxes; synthesized outputs are shown in dashed boxes. On the right, the Queue module’s specifications and implementation are expanded; the other modules would have similar details.

TicketVendor module, denoted by the yellow box labeled \( \phi_{\text{imp}} \) in Fig. 1, might constrain issueTicket to dequeue a buyer from the underlying Queue module, but only if soldOut (a TicketVendor operation) is false.

**Module 2:** Specifications of the behaviors of the Queue operations in terms of the exposed Stack operations (and possibly other Queue operations). For example, the implementation-specific specification for the Queue module (\( \phi_{\text{imp}} \)), shown in Fig. 1, contains, among others, constraints that state that (i) if the first stack \( st_{\text{in}} \) is empty, so is the second stack \( st_{\text{out}} \), (ii) enqueuing 1 on an empty queue and then retrieving the front of the queue yields 1.

**Module 1:** Specifications of the behaviors of the Stack operations in terms of the programming language’s linked-list operations (and possibly other Stack operations). For example, the implementation-specific specification of the Stack module (\( \phi_{\text{imp}} \)) might specify that push adds an element on the front of the stack’s underlying linked list.

A user must also specify a search space of possible implementations. In MoSSKit, this is done using a SKETCH file.

### C. The Output of Modular TicketVendor Synthesis

Using the MoSS framework, we synthesize three module implementations: the TicketVendor module implementation, which satisfies \( \phi_{\text{imp}} \) (and uses Queue); the Queue module implementation, which satisfies \( \phi_{\text{imp}} \) (and uses Stack); and the Stack module implementation, which satisfies \( \phi_{\text{imp}} \) (and uses lists). However, to synthesize the TicketVendor module implementation, we need an implementation-agnostic specification of Queue, denoted by \( \phi_{\text{sem}} \). The same can be said for the Queue module implementation, for which we need an implementation-agnostic specification of Stack, denoted by \( \phi_{\text{sem}} \). The user could write \( \phi_{\text{Queue}} \) and \( \phi_{\text{Stack}} \) manually, but it is more convenient to synthesize these specifications from the Queue and Stack module implementations, respectively. The MoSS methodology is to start with the bottom-most module and work upward, alternately applying two synthesis procedures: first synthesizing the implementation of a module \( M \) and then synthesizing \( M \)’s implementation-agnostic specification \( \phi_{\text{Msem}} \), which gets exposed to the next higher module.

For the modular TicketVendor-synthesis problem, we start with Stack, the bottommost module, and synthesize a Stack module implementation—a set of \( \phi_{\text{List}} \) programs—that satisfies the implementation-specific specification \( \phi_{\text{Stack}} \). (In MoSSKit, this step is done using program sketching and the tool JLIBSKETCH [14].) This step is depicted in Fig. 1 as the Implementation Synthesis problem in the Stack module. We then switch to the Specification Synthesis problem for Stack, and synthesize \( \phi_{\text{sem}} \), an implementation-agnostic specification of Stack. (In MoSSKit, this step is done by providing a grammar of possible properties and by using the tool SPYRO [15].) For the Stack module, the resultant \( \phi_{\text{sem}} \) is the conjunction of the equalities shown at 1 in Fig. 1.

Using \( \phi_{\text{Stack}} \) (1), together with the implementation-specific specification \( \phi_{\text{imp}} \) (2), we now synthesize the Queue module implementation (3)—a set of \( \phi_{\text{List}} \) programs—and the implementation-agnostic specification \( \phi_{\text{sem}} \) (4) via the same two-step process.

Finally, using \( \phi_{\text{Queue}} \) and the implementation-specific specification \( \phi_{\text{TicketVendor}} \), we synthesize the TicketVendor module implementation. (If needed by a further client, we would then synthesize the implementation-agnostic specification \( \phi_{\text{TicketVendor}} \)) Thus, the last output of the synthesis procedure, shown in Fig. 1, consists of implementations of Stack, Queue, and TicketVendor, and the implementation-agnostic specifications \( \phi_{\text{Stack}} \) and \( \phi_{\text{Queue}} \).

### D. Benefits of Modular System Synthesis

At some point, we might want to decide to modify the implementation of the Queue module to use directly the linked-list primitives provided by the language (shown in Fig. 2). Information hiding allows us to do so in a compartmentalized way—i.e., by only changing the specific Queue module. Importantly, the module’s interface, composed of the function

---

2Technically, List is part of the programming language; however, so that all sub-problems have the same form, we assume—as shown in Fig. 1—that we also have available an implementation-agnostic specification of List, denoted by \( \phi_{\text{List}} \). In our evaluation, we synthesize \( \phi_{\text{List}} \) automatically.
symbols in \( P[\text{Queue}] \) and its implementation-agnostic specification \( \varphi_{\text{sem}}^{\text{Queue}} \), does not change when the implementation of the Queue module changes. Because this interface is what the TicketVendor module was synthesized with respect to, changes to the Queue implementation are not visible to TicketVendor.

III. MODULAR SYSTEM DESIGN

In this section, we formally define modular system design and the corresponding specification mechanisms. A system is organized in modules, and each module exports a module interface \( MI \) and a specification \( \varphi_{\text{sem}}^{MI} \) of the semantics of the module interface. Both \( MI \) and \( \varphi_{\text{sem}}^{MI} \) hide the module’s implementation. A module’s implementation can also have a set of private functions \( PF \), which can only be used within the module. A program is constructed by stacking layers of such modules. For instance, the example in Fig. 1 has three modules: Stack, Queue, and TicketVendor. (None of those modules have private functions.)

In the following, we assume a programming language \( P \) (e.g., C with its core libraries), and use \( P[MI] \) to denote \( P \) extended with the functions exposed by module \( MI \).

Definition 1 (Modular System Design): A system is implemented modularly if it is partitioned into disjoint sets of functions \( PF_1, MI_1, PF_2, MI_2, \ldots, PF_n, MI_n \), such that for each \( f \in PF_i \cup MI_i, f \) is implemented using \( P[MI_{i-1} \cup PF_i \cup MI_i] \) —i.e., \( f \) only uses operations in \( P \), and calls to functions in the interface exported from layer \( i-1 \), to private functions of layer \( i \), and to functions in the interface exported from layer \( i \).

To reduce notational clutter, we will ignore private functions, and only discuss the functions in module interfaces.

As we saw in §II, we need to abide by the principle of information hiding—i.e., changing the implementations of any function in \( MI_{i-1} \) should not require changing the implementations of functions in \( MI_i \). With this principle in mind, we now describe the different natures of the specification for the module implementation at a given layer \( i \) (§III-A) and the specification exposed to layer \( i+1 \) (§III-B).

1In general, the structure of the dependencies among layers can form a directed acyclic graph. However, to reduce notational clutter, throughout the paper we assume that the layers have a strict linear order.

A. Implementation-specific Specifications

When synthesizing specific implementations of the functions \( MI_i \) at layer \( i \), the specifications are allowed to use symbols in \( P[MI_{i-1} \cup MI_i] \) —i.e., the specification can refer to the functions we are specifying and to the ones in the interface exported from the previous layer—as well as implementation-specific details from layer \( i \) (e.g., data-structure declarations).

Definition 2: An implementation-specific specification for a set of functions \( MI_i \) at layer \( i \) is a predicate \( \varphi_{\text{imp}}^{MI_i} \) that only uses symbols in \( P[MI_{i-1} \cup MI_i] \).

Example 1: In the implementation-specific specification of Queue from Fig. 1, where Queue is implemented using two Stacks, one of the properties is as follows:

\[
\text{isEmptyQ}(q) \iff \text{isEmptyS}(q.st_{in}) \land \text{isEmptyS}(q.st_{out}).
\]

For the version from Fig. 2, where Queue is implemented using a List, the analogous property is

\[
\text{isEmptyQ}(q) \iff \text{isEmptyL}(q.l).
\]

A specification might also contain a set of examples, e.g., \( \text{front}(\text{enq}(\text{emptyQ}, 1)) = 1 \) and \( \text{front}(\text{enq}(\text{emptyQ}, 2)) = 1 \).

B. Implementation-agnostic Specifications

While implementation-specific details are needed to converge on an implementation with which the programmer is happy, when exposing the specification of \( MI_i \) at layer \( i \), to abide to the principle of information hiding, one cannot provide specifications that involve function symbols in \( P[MI_{i-1} \cup MI_i] \), but only those in \( P[MI_i] \).

Definition 3: An implementation-agnostic specification for a set of functions \( MI_i \) at layer \( i \) is a predicate \( \varphi_{\text{sem}}^{MI_i} \) that only uses symbols in \( P[MI_i] \).

Example 2: Because of the vocabulary restrictions imposed by Def. 3, it is natural for implementation-agnostic specifications to take the form of algebraic specifications [7], [9], [10], [13], [23]. For instance, for the Queue module, the conjunction of the following equalities is an implementation-agnostic specification \( \varphi_{\text{sem}}^{\text{Queue}} \) for Queue:

\[
\begin{align*}
\text{isEmptyQ}(\text{emptyQ}) &= T & \text{isEmptyQ}(\text{enq}(q, x)) &= \bot \\
\text{sizeQ}(\text{emptyQ}) &= 0 & \text{sizeQ}(\text{enq}(q, x)) &= \text{sizeQ}(q) + 1 \\
\text{front}(\text{enq}(q, x)) &= \text{ite}(\text{isEmptyQ}(q), x, \text{front}(q)) \\
\text{deq}(\text{enq}(q, x)) &= \text{ite}(\text{isEmptyQ}(q), q, \text{deq}(\text{enq}(q, x)))
\end{align*}
\]

Note that Eq. (1) serves as \( \varphi_{\text{Queue}} \) both for the version of Queue from Fig. 1, where Queue is implemented using two Stacks, and for the version of Queue from Fig. 2, where Queue is implemented using a List.

IV. SYNTHESIS IN MODULAR SYSTEM SYNTHESIS

In this section, we define the implementation-synthesis (§IV-A) and specification-synthesis (§IV-B) problems that enable our scheme for modular system synthesis.
A. Synthesis of Implementations

The obvious place in which synthesis can be helpful is in synthesizing the implementations of the various functions at each layer from their implementation-specific specifications. For example, in Fig. 1, an implementation of Queue (the function enq is shown in the second box on the right) is synthesized from the implementation-agnostic specification \( \varphi_{\text{sem}} \) of Stack, and an implementation-specific specification \( \varphi_{\text{imp}} \) that is allowed to talk about how the two Stacks used to implement a Queue are manipulated (e.g., is\( \text{Empty} \)(st\(_{\text{out}}\)) \( \rightarrow \) is\( \text{Empty} \)(st\(_{\text{in}}\)).

Definition 4 (Implementation synthesis): For module interface \( M_i \), the implementation-synthesis problem is a triple \((S_i, \varphi_{\text{sem}}, \varphi_{\text{imp}})\), where

- \( S_i \) is the set of possible implementations we can use for \( M_i \) (every program in \( S_i \) uses only symbols in \( P[M_{i-1} \cup M_i] \)).
- \( \varphi_{\text{sem}}^{M_i} \) is an implementation-agnostic specification of the module-interface functions in \( M_{i-1} \).
- \( \varphi_{\text{imp}}^{M_i} \) is an implementation-specific specification that uses only symbols in \( P[M_{i-1} \cup M_i] \).

A solution to the implementation-synthesis problem is an implementation of \( M_i \) in \( S_i \) that satisfies \( \varphi_{\text{imp}}^{M_i} \).

This particular form of synthesis where one draws a program from a search space to match a specification is fairly standard in the literature. However, we observe that a particular aspect of modular system design makes most synthesis approaches inadequate—i.e., the specification \( \varphi_{\text{sem}}^{M_i} \) can talk about functions in \( M_{i-1} \) only in an implementation-agnostic way. For example, when synthesizing functions in Queue, we do not have direct access to a stack implementation—i.e., we cannot actually execute the implementation. Instead, we have access to the semantics of Stack through implementation-agnostic properties such as \( \text{is\emptyset}(\text{push}(st, x)) = \perp \).

We are aware of only one tool, JLibSketch, that can perform synthesis with algebraic specifications [14], and we use it in our evaluation. In JLibSketch, one provides \( S_i \) as a program sketch (i.e., a program with integer holes that need to be synthesized), \( \varphi_{\text{sem}}^{M_{i-1}} \) as a set of rewrite rules over the functions in \( M_{i-1} \), and \( \varphi_{\text{imp}}^{M_i} \) as a set of assertions.

B. Synthesis of Implementation-agnostic Specifications

Because the implementation of layer \( i-1 \) is hidden when performing synthesis at layer \( i \), the user has to somehow come up with implementation-agnostic specifications like the ones shown in Fig. 1. Our next observation is that such specifications can also be synthesized! With this observation, modular system design becomes a fairly automatic business where the programmer mostly has to decide how to structure modules and provide implementation-specific specifications and search spaces (typically as regular-tree grammars [3]).

In Fig. 1, the implementation-agnostic specification \( \varphi_{\text{sem}}^{\text{Queue}} \) of Queue is synthesized from the Queue implementation. (The same \( \varphi_{\text{imp}}^{\text{Queue}} \), or one equivalent to it, is synthesized from the alternative Queue implementation of Fig. 2.)

Definition 5 (Specification synthesis): For module interface \( M_i \), a specification-synthesis problem is a pair \((F_i, \Phi_i)\) where

- \( F_i \) is a set of programs, written in \( P[M_{i-1} \cup M_i] \), that is a concrete implementation of \( M_i \).
- \( \Phi_i \) is the set of possible properties we can use for \( \varphi_{\text{sem}}^{M_i} \) (every property in \( \Phi_i \) uses only symbols in \( P[M_{i-1}] \)).

A solution to the specification-synthesis problem is a set of properties \( \varphi_{\text{sem}}^{M_i} \subseteq \Phi_i \) such that for every \( \alpha \in \varphi_{\text{sem}}^{M_i} \), a concrete implementation of \( \alpha \) can be synthesized in any order.

Discussion. When the goal is to build a system structured in a modular fashion, modular system synthesis enables defining “small” synthesis problems of similar nature that concern only a single module’s implementation.

While implementation-agnostic specifications can be synthesized via the synthesis problem defined in Def. 5, one should be aware that there is additional flexibility to be gained if one is willing to write implementation-agnostic specifications manually. In particular, if all of the implementation-agnostic specifications are synthesized, then it is necessary to create the system bottom-up, synthesizing the module implementations in the order \( M_1, M_2, \ldots, M_n \) (interleaved with the synthesis of \( \varphi_{\text{imp}}^{M_1}, \varphi_{\text{imp}}^{M_2}, \ldots, \varphi_{\text{imp}}^{M_n} \)). In contrast, when the user is willing to write the implementation-agnostic specifications manually (in addition to the implementation-specific specifications \( \varphi_{\text{imp}}^{M_i} \)), then the module implementations for \( M_1, M_2, \ldots, M_n \) can be synthesized in any order.

V. IMPLEMENTATION AND CASE-STUDY EVALUATION

We carried out case studies of MoSS for the simple three-layer system that has been used as a running example and for some of the modular-synthesis problems presented in the paper that introduced JLibSketch [14].

A. Implementation

Our implementation, called MoSSKit, uses JLibSketch [14] to synthesize the implementation code for each layer \( k \) (from the implementation-specific specification for layer \( k \))
and SPYRO [15] to synthesize the implementation-agnostic specification for use at layer \( k + 1 \).

JLibSketch is a program-synthesis tool for Java that allows libraries to be described with collections of algebraic specifications. Similar to its popular C counterpart Sketch [22], JLibSketch allows one to write programs with holes and assertions, and then tries to find integer values for the holes that cause all assertions to hold. Each specification is a rewrite rule of the form \( \phi \Rightarrow L \). For instance, the rule

\[
\text{is_empty(l).}
\]

\[
\text{is_empty(l, is_empty(ret));}
\]

\[
\text{if (is_empty_ret) }
\]

\[
\text{ret_list.hd = val;}
\]

\[
\text{nil(ret.tl);}
\]

\[
\text{else }
\]

\[
\text{ret_list.hd = l.hd;}
\]

\[
\text{snoc(l.tl, val, ret.tl);}
\]

\[
\text{if (is_empty_ret) }
\]

\[
\text{snoc_out(l, I) = cons(I, L);}
\]

\[
\text{if (is_empty(I) || is_empty(l))}
\]

\[
\text{GUARD \rightarrow true}
\]

\[
\text{RHS \rightarrow equal_list(snoc_out, L)}
\]

\[
\text{I \rightarrow v1 | v2}
\]

\[
\text{L \rightarrow l | nil()}
\]

\[
\text{snoc(l, I) | cons(I, l)}
\]

\[
\text{snoc_out = cons(v1, snoc(l, v2));}
\]

\[
\text{snoc_out = cons(v1, snoc(v1, v2))}
\]

In this case, the only expression for GUARD that succeeds is \( T \), and the property synthesized is

\[
\text{snoc_out = cons(v1, snoc(v1, v2))}
\]

SPYRO—i.e., the synthesized implementations and specifications are sound up to a bound. Despite this limitation, the authors of JLibSketch and SPYRO have shown that these tools typically do not return unsound results in practice. §V-E provides a detailed discussion of the limitations of MOSS and MoSSKIt.

### B. Ticket-vendor Case Study

Our first benchmark is the ticket-vending application described throughout the paper. Our goal is to synthesize the four module implementations in Fig. 1 (except the bottom one), as well as the specification of each module that needs to be exposed to a higher-level module.

When synthesizing specifications, due to the scalability limitations of SPYRO, we called SPYRO multiple times with different smaller grammars instead of providing one big grammar of all possible properties of each module. In each call to SPYRO, we provided a grammar in which we fixed a left-hand-side expression of an equality predicate, and asked SPYRO to search for a right-hand-side expression for the equality. We allowed the right-hand-side expression to contain a conditional where the guard can be selected from the outputs of Boolean operators in the module, their negation, or constants. For instance, Figures 3 and 4 illustrate two inputs provided to SPYRO to solve the specification-synthesis problem for List:

(i) a program describing the implementation of List (Fig. 3), and
(ii) a grammar describing the set of possible properties (Fig. 4).

Because we wanted to use the synthesized equalities as input to JLibSketch when synthesizing the implementation
of the next higher-level module, we provided grammars of
equalities that avoided generating cyclic rewrite rules. We
addressed this issue by limiting the search space for the
right-hand-side expression. The function symbols permitted in the
generators from which JLisketch can pick an expression. For these
generators, expressions can be variables or single function calls to functions
of the appropriate type—e.g., genStack2 can generate expressions such as
st_in, st_out, st_in.pop(), st_out.pop()), etc.

To illustrate some of the properties synthesized by
MOSSKIT (that are not shown in Fig. 1) the complete set of
equalities in the implementation-agnostic specification ϕ \_list \_sem synthesized by Spyro is the following:

```
public void enq(int x) {
    Stack st_in = this.st_in;
    Stack st_out = this.st_out;
    assume !st_out.isEmpty() || st_in.isEmpty();
    if (genGuard(st_in, st_out)) {
        st_in = genStack2(st_in, st_out, x);
        st_out = genStack2(st_in, st_out, x);
    }
    st_in = genStack2(st_in, st_out, x);
}
```

When considering the cumulative time taken to synthesize
the algebraic specification of each module, Spyro took 41
seconds for \( \varphi \_\text{list} \_\text{sem} \) (longest-taking property 7 seconds), 34
seconds for \( \varphi \_\text{stack} \_\text{sem} \) (longest-taking property 7 seconds), and
44 seconds for \( \varphi \_\text{queue} \_\text{sem} \) (longest-taking property 13 seconds).

We used JLisketch to synthesize implementations of the
modules. In addition to the implementation-agnostic speciﬁcation
of the module below the one we were trying to synthesize, we provided an implementation-speciﬁcation of the module to be synthesized. For example, the \( \varphi \_\text{stack} \_\text{sem} \) speciﬁcation involved JLisketch code with 17 assertions, and the following examples are an excerpt from the \( \varphi \_\text{stack} \_\text{sem} \) speciﬁcation \((x, y, z)\) are universally quantiﬁed integers

that are allowed to be in the range 0 to 10):

\[
\begin{align*}
\text{top}(\text{push}(\text{emptyS}, x)) &= x \\
\text{top}(\text{push}(\text{emptyS}, x), y) &= y \\
\text{sizeS}(\text{emptyS}) &= 0 \\
\text{sizeS}(\text{push}(\text{emptyS}, x)) &= 1
\end{align*}
\]

Besides the assertions, we provided JLisketch with a
fairly complete sketch of the structure of the implementation:
we provided loops and branching structures, and only asked
JLisketch to synthesize basic statements and expressions. For example, the sketch provided for the operation enq of
module Queue = (st_in : Stack, st_out : Stack) is shown in Fig. 5. This sketch of enq of module Queue uses two
Stacks, st_in, which stores elements in the rear part of the
queue, and st_out, which stores elements in the front part of
the queue. Stack st_in holds the rearmost element on top, and Stack st_out stores the frontmost element on top. To make
the front operation more eﬃcient, we decided to make sure
that the frontmost element is always at the top of st_out. This
implementation decision is expressed as assertions in lines 5
and 15 constituting an implementation-speciﬁc speciﬁcation
\( \varphi \_\text{sem} \) Queue , shown as \( \ast \) in Fig. 1.

Afterward, based on the implementation synthesized by
JLisketch, Spyro was able to solve each Queue speciﬁcation-synthesis problem within 40 seconds, yielding the following implementation-agnostic speciﬁcation
\( \varphi \_\text{sem} \) Queue :

\[
\begin{align*}
\text{isEmptyQ}(\text{emptyQ}) &= \top \\
\text{isEmptyQ}(\text{enq}(q, i)) &= \bot \\
\text{sizeQ}(\text{emptyQ}) &= 0 \\
\text{sizeQ}(\text{enq}(q, i)) &= \text{sizeQ}(q) + 1 \\
\text{isEmptyQ}(q) &= \text{sizeQ}(q) + i \\
\text{isEmptyQ}(q) &= \text{front}(\text{enq}(q, i)) = \text{front}(q) \\
\text{isEmptyQ}(q) &= \text{deq}(\text{enq}(q, i)) = q \\
\text{isEmptyQ}(q) &= \text{deq}(\text{enq}(q, i)) = \text{enq}(\text{deq}(q, i))
\end{align*}
\]

A TicketVendor is implemented using a Queue, which
stores the id numbers of clients who have reserved tickets.
Each issued ticket contains the id of the buyer. The
implementation-speciﬁc speciﬁcation \( \varphi \_\text{sem} \) TicketVendor consisted
of JLisketch code with 24 assertions, and contains multiple
eamples, such as the following (again, \( x \) and \( y \) are universally
quantiﬁed integers that are allowed to be in the range 0 to 10):

\[
\begin{align*}
\text{numTicketsRem}(\text{prepSales}(2)) &= 2 \\
\text{numWaiting}(\text{prepSales}(2)) &= 0 \\
\text{numWaiting}(\text{resTicket}(\text{prepSales}(2), x)) &= 1 \\
\text{issueTicket}(\text{resTicket}(\text{prepSales}(2), x)).\text{owner} &= x
\end{align*}
\]

Again, we provided JLisketch with a fairly complete
sketch of the program structure, and JLisketch was able to synthesize the implementations of all the
TicketVendor functions within 10 seconds. For example, the function \( \text{prepSales} \) for TicketVendor = (numTicket : int, queue : Queue) was synthesized as \( \text{prepsales}(n : \text{int}) := \langle n, \text{emptyQ} \rangle \).

We compared the time needed to synthesize each module
from the algebraic speciﬁcation of the previous module to the
time needed to synthesize using the implementation of
all previous modules. Synthesizing Stack from the speciﬁcation \( \varphi \_\text{sem} \) took 3 seconds instead of the 2 seconds
needed when the implementation of List was provided. Synthesizing Queue from the speciﬁcation \( \varphi \_\text{sem} \) Stack took 188
seconds instead of the 799 seconds needed when the concrete implementations of \( \text{Stack} \) and \( \text{List} \) were provided. Synthesizing \( \text{TicketVendor} \) from the specification \( \varphi_{\text{TicketVendor}} \) took 7 seconds, but \texttt{JLIBSKETCH} crashed when the concrete implementations of \( \text{Queue} \), \( \text{Stack} \) and \( \text{List} \) were provided.

**Key finding:** This experiment shows that modular synthesis takes 1-5 minutes per module, whereas the time taken to synthesize a module from the underlying module implementation grows with the number of modules—to the point where synthesis is unsuccessful with existing tools.

As discussed in §II-D, we also synthesized an implementation of \( \text{Queue} \) that uses \( \text{List} \) instead of two \( \text{Stacks} \). The \( \text{List} \) holds the oldest element of the \( \text{Queue} \) at its head. The implementation-specific specification \( \varphi_{\text{Queue as List}} \) consisted of \texttt{JLIBSKETCH} code with 19 assertions, including examples similar to those shown at \( \bullet \) in Fig. 2. We used \texttt{JLIBSKETCH} to verify whether the specification \( \varphi_{\text{Queue}} \) still held true for the new implementation. Because it did (confirmation took <1 second), \( \text{TicketVendor} \) does not need to be changed to use the \( \text{Queue} \) implementation.

**C. Case Studies from Mariano et al. [14]**

Our second set of benchmarks is collected from the paper that introduced synthesis from algebraic specifications via \texttt{JLIBSKETCH} [14]. In that work, Mariano et al. used a number of benchmarks that involve two modules—e.g., synthesizing a backend cryptographic component for a tool that brings NuCypher to Apache Kafka, using \texttt{ArrayList} and \texttt{HashMap} as underlying modules. The goal of their paper was to show that in \texttt{JLIBSKETCH} it was easier/faster to synthesize the module at layer 1 when the module of layer 0 was exposed through an algebraic specification (rather than a concrete implementation). The current implementation of \texttt{MOSSKIT} does not support strings, so we used only the benchmarks for which the algebraic specifications for the layer-0 module (i) did not use \texttt{string} operations, and (ii) did not use auxiliary functions that were not in the signature of the module. In total, we considered four layer-0 modules: \texttt{ArrayList}, \texttt{TreeSet}, \texttt{HashSet}, and \texttt{HashMap}. Each \texttt{JLIBSKETCH} benchmark consisted of (i) an algebraic specification of the layer-0 module (written by hand), (ii) a \texttt{SKETCH}-like specification of the layer-1 module, and (iii) a mock implementation of the layer-0 module—i.e., a simplified implementation that mimics the module’s intended behavior (e.g., \texttt{HashSet} is implemented using an array). The mock is not needed by \texttt{JLIBSKETCH}, but allowed Mariano et al. to compare synthesis-from-algebraic-specifications against synthesis-from-mocks [14, §5].

We used these items in a different manner from the \texttt{JLIBSKETCH} experiments. From just the mock implementation of layer 0, we asked \texttt{MOSSKIT} to synthesize a most-precise algebraic specification, which we compared with the algebraic specification manually written by Mariano et al. From that algebraic specification and the \texttt{SKETCH}-like specification of the layer-1 module, we asked \texttt{MOSSKIT} to synthesize the implementation of layer 1. (The second step essentially replicated the algebraic-synthesis part of the \texttt{JLIBSKETCH} experiments.) For the layer-0 synthesis step of each benchmark, we synthesized algebraic specifications using grammars similar to the ones used in §V-B.

When considering the time taken to synthesize the entire algebraic specification of each module, \texttt{SPYRO} took 626 seconds for \( \varphi_{\text{ArrayList}} \), 54 seconds for \( \varphi_{\text{HashSet}} \), and 1,732 seconds for \( \varphi_{\text{HashMap}} \). Because mock implementations are simplified versions of actual implementations, the mock implementation of \texttt{TreeSet} is identical to the mock implementation of \texttt{HashSet}—i.e., they both represent sets as arrays. Furthermore, the two implementations have the same algebraic specifications—i.e., \( \varphi_{\text{TreeSet}} = \varphi_{\text{HashSet}} \)—which can thus be synthesized in the same amount of time.

**Key finding:** For all but two benchmarks, the \( \mathcal{L} \)-conjunctions synthesized by \texttt{MOSSKIT} were equivalent to the algebraic properties manually written by Mariano et al. For the mock implementation of \texttt{HashMap} and \texttt{ArrayList} provided in \texttt{JLIBSKETCH}, for specific grammars, \texttt{MOSSKIT} synthesized empty \( \mathcal{L} \)-conjunctions (i.e., the predicate true) instead of the algebraic specifications provided by Mariano et al.—i.e., \( k_1 = k_2 \Rightarrow \text{get}(\text{put}(m, k_1, v), k_2) = v \) and \( i = j \Rightarrow \text{get}(\text{set}(l, i, v), j) = v \), for \texttt{HashMap} and \texttt{ArrayList}, respectively. Upon further inspection, we discovered that \texttt{JLIBSKETCH}’s mock implementation of \texttt{HashMap} was incorrect, and did not satisfy the specification that Mariano et al. gave, due to an incorrect handling of hash collision! After fixing the bug in the mock implementation of \texttt{HashMap}, we were able to synthesize the expected algebraic specification. However, when inspecting the implementation of \texttt{ArrayList}, we found that for this benchmark the implementation was correct but the algebraic specification provided by Mariano et al. was incorrect! After modifying the grammar, we could synthesize the correct algebraic specification \( (i = j) \land (0 \leq i) \land (i \leq \text{sizeL}(l)) \Rightarrow \text{get}(\text{set}(l, i, v), j) = v \). However, this modification revealed a bug in one of the implementations of \texttt{HashMap} that Mariano et al. had synthesized from the earlier erroneous specification! We discuss this finding further in the next section.

This finding illustrates how modular system synthesis can help to identify and avoid bugs in module implementations.

**D. Additional Case Studies Based on Mariano et al. [14]**

We noticed that the \texttt{JLIBSKETCH} benchmarks provided an opportunity to build a more complicated benchmark that involved 3 modules (instead of 2). In particular, two of the benchmarks involved synthesizing the implementation of a (layer-1) \texttt{HashMap} module from a (layer-0) algebraic specification of \texttt{ArrayList}. (The two benchmarks synthesized different implementations that handled collisions differently and we refer to the corresponding modules as \texttt{HashMap1} and \texttt{HashMap2}.) The third benchmark involved synthesizing the implementation of a (layer-2) \texttt{Kafka} from a (layer-1) algebraic specification of \texttt{HashMap}. Thus, we built two 3-layer benchmarks in which the goal was to synthesize \texttt{Kafka} using an implementation of \texttt{HashMap} that used an implementation of \texttt{ArrayList}. For us, each 3-layer benchmark involved four
synthesis problems: (1) the algebraic specification $\varphi_{\text{ArrayList}}$ of ArrayList (from the mock); (2) the implementation of either HashMap1 or HashMap2; (3) the algebraic specification of HashMap; and (4) the implementation of Kafka (this part was already synthesized in [14]).

As discussed in the previous section, we identified a bug in the specification $\varphi_{\text{ArrayList}}$ manually provided by Mariano et al., and were able to use MoSSKIt to synthesize a correct algebraic specification—i.e., step (1). For step (2), the implementation synthesized by Mariano et al. for HashMap2 was still correct, and we could also use MoSSKIt to synthesize it from the corrected specification $\varphi_{\text{ArrayList}}$. However, the implementation of HashMap1 synthesized by JLIBSKETCH was incorrect because it depended on the original, erroneous specification $\varphi_{\text{ArrayList}}$ for ArrayList—(1) put could store values to negative indices; and (2) get could search key from incorrect index after rehashing. We manually changed the implementation of the rehashing function in the sketch of HashMap1 to fix the bug, but the change was large enough that we did not attempt to rewrite the program sketch needed to synthesize this specification (i.e., we manually wrote the implementation of HashMap1 instead of synthesizing it). Synthesis problem (3) is at the heart of handling a multi-module system in a modular fashion: we used MoSSKIt to synthesize algebraic specifications of HashMap1 and HashMap2—in each case, giving MoSSKIt access to the (correct) implementations of HashMap1 and HashMap2 and the (correct) algebraic specification of ArrayList (but not an implementation of ArrayList).

Key finding: MoSSKIt failed to synthesize the same algebraic specification we had obtained for HashMap in §V-C when attempting to synthesize a specification for HashMap1 and HashMap2. When inspecting the synthesized properties, we realized that the algebraic specification $\varphi_{\text{ArrayList}}$ exposed by ArrayList still had a problem! In particular, $\varphi_{\text{ArrayList}}$ was too weak to prove the algebraic specifications needed by HashMap1 and HashMap2—i.e., $\varphi_{\text{ArrayList}}$ did not characterize properties that were needed by HashMap1 and HashMap2 to satisfy the algebraic specification $\varphi_{\text{HashMap}}$. We used Sketch to itself produce a violation of the algebraic specification $\varphi_{\text{sem HashMap}}$ for HashMap1 under the weaker assumption that ArrayList only satisfied the specification $\varphi_{\text{ArrayList}}$, and used the violations generated by Sketch to identify what properties we needed to add to strengthen $\varphi_{\text{ArrayList}}$. In particular, sizeL(ensureCapacity(l, n)) = sizeL(l) and get(ensureCapacity(l, n), i) = get(l, i) were added to describe the behavior of ensureCapacity. We were then able to modify the grammar used to synthesize algebraic specifications for ArrayList and synthesize the missing property. After obtaining $\varphi_{\text{ArrayList}}$, we successfully synthesized the full algebraic specification for HashMap2 (i.e., $\varphi_{\text{sem HashMap2}}$) and most of the algebraic specification for HashMap1. Because the corrected implementation of HashMap1 was particularly complicated—e.g., each call to put requires rehashing when the load factor is greater than a predefined value—MoSSKIt timed out while synthesizing every property, with the exception of the property get(emptyMap, k) = err.

This finding illustrates how modular system synthesis can help identify when module specifications are not strong enough to characterize the behavior of other modules.

E. Limitations of MoSSKIt

JLIBSKETCH and SPYRO represent the algebraic specifications of modules as rewrite rules for algebraic datatypes (ADTs). Reasoning about ADTs is a challenging problem, and to the best of our knowledge, Sketch and JLIBSKETCH are only frameworks capable of handling problems involving ADTs effectively. Therefore, MoSSKIt uses them as the underlying solver and inherits limitations of Sketch.

The primary limitation of MoSSKIt is its bounded soundness guarantee. Sketch ensures soundness only for a bounded number of loop/recursion unrollings, and bounded input sizes. Verifying the unbounded correctness of the synthesized programs poses a significant challenge, as semantics of lower-level modules are represented as rewrite rules on ADTs. As a future direction, we plan to integrate MoSSKIt with verifiers such as Dafny to perform full verification, as was done in [15] for the properties synthesized by SPYRO. However, it is worth noting that MoSSKIt has already been useful in finding bugs in existing implementations: specification synthesis has helped find implementation errors in the case studies of Mariano et al. [14], as demonstrated in §V-C and §V-D.

Although the case studies in §V-B and reference [14] show satisfactory performance of Sketch for most problems, scalability issues persist. In particular, unrolling nested loops significantly increases the number of holes of the Sketch problem, which increases the problem’s difficulty.

Besides the limitations inherited from Sketch, MoSS has a specific requirement for the system’s modular structure, which should be a directed acyclic graph (DAG)—i.e., the implementation-agnostic specifications of all dependent modules must be provided to synthesize a particular module. MoSS addresses the challenges in writing accurate specifications by using the synthesis of implementation-agnostic specifications. However, in this approach one needs to synthesize all dependent modules and their specifications before attempting to synthesize a new module. Alternatively, to synthesize higher-level modules without the lower-level implementations, the user can manually supply the implementation-agnostic specifications of the lower-level modules.

VI. Related Work

A problem related to ours is that of component-based synthesis (CBS), where the goal is assembling pre-existing components/APIs to generate more complex programs. Many existing approaches for solving CBS problems scale reasonably well [5], [18], [20], but require the individual components to be executable. In our setting, this approach is not possible because the details of lower-level components (e.g., how a Stack is implemented) need not be observable.

A few tools have abstracted components and modules using specifications. JLIBSKETCH [14] uses algebraic properties to
represent the semantics of modules and is a key component of our implementation. (CL)S [2] and APIphany [8] use types to represent the behavior of components and can be used in tandem with specialized type-directed synthesizers. The key differences between our work and these tools is that MOSS provides two well-defined synthesis primitives that support composing multiple modules, rather than synthesizing just one implementation for one module. Furthermore, the aforementioned types are limited in how they can represent relations between multiple components in an implementation-agnostic way, thus making us opt for algebraic specifications.

Many synthesis tools perform some kind of “compositional” synthesis by breaking an input specification into sub-specifications that are used to separately synthesize sub-components of a target program [1], [17]. This notion of “compositionality” is orthogonal to ours, and is more of a divide-and-conquer approach to solving individual synthesis problems. MOSS can make use of such a divide-and-conquer approach when synthesizing a module’s implementation.

For the task of synthesizing an algebraic specification, MoSSKit uses SYPYRO. Besides SYPYRO, there are a number of works about discovering specifications from code, based on both static techniques [6], [21] and dynamic techniques [4], [11]. The static approaches mostly target predicates involving individual functions (instead of algebraic properties and equalities involving multiple functions). The dynamic techniques are flexible and can identify algebraic specifications (e.g., for Java container classes [11]), but require some “bootsfitting” inputs, and only guarantee soundness with respect to behaviors that are covered by the tests that the inputs exercise.

VII. CONCLUSION

**Conceptual contributions.** At the conceptual level, this paper contributes both a framework and a new way to think about program synthesis that opens many research directions. Specifically, the paper introduces MoSS, a framework for using synthesis to perform modular system synthesis. The main contribution of this paper is not an immediate solution to the modular-synthesis problem, but rather the identification of two key synthesis primitives that are required to realize MOSS in practice: 1) synthesis from an implementation-agnostic specification, and 2) synthesis of an implementation-agnostic specification. While our tool implements both of these primitives using tools based on SKETCH (thus inheriting its limitations), an interesting research direction is whether other synthesis approaches (enumeration, CEGIS, etc.) can be extended to handle our synthesis problems, perhaps by leveraging the popular egg framework [24] which allows one to reason about equivalence of terms with respect to a term-rewriting system—i.e., our algebraic specifications.

**Experimental Contributions.** We created MoSSKit, a proof-of-concept implementation of MOSS based on two existing program-synthesis tools: JLibSketch [14], a program-sketching tool that supports algebraic specifications, and SYPYRO [15], a tool for synthesizing precise specifications from code. The case studies carried out with MoSSKit show that (i) modular synthesis is faster than monolithic synthesis, and (ii) performing synthesis for both implementations and specifications of the modules can prevent subtle bugs.

**ACKNOWLEDGEMENT**

Supported, in part, by a Microsoft Faculty Fellowship, a gift from Rajiv and Ritu Batra; by ONR under grant N00014-17-1-2889; and by NSF under grants CCF-1750965, 1763871, 1918211, 2023222, 2211968, 2212558.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

**REFERENCES**


