Formal Methods in Computer-Aided Design 2023

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

> Can one address the scalability problem of program synthesis by exploiting compositionality, so that (i) larger programs are synthesized from smaller programs, and (ii) each "larger-created-from-smaller" synthesis sub-problem is of a similar nature, so that the essence of each sub-problem (and the size of each search space) has roughly the same character?

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

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_34

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 SPYRO [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].[1]
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:

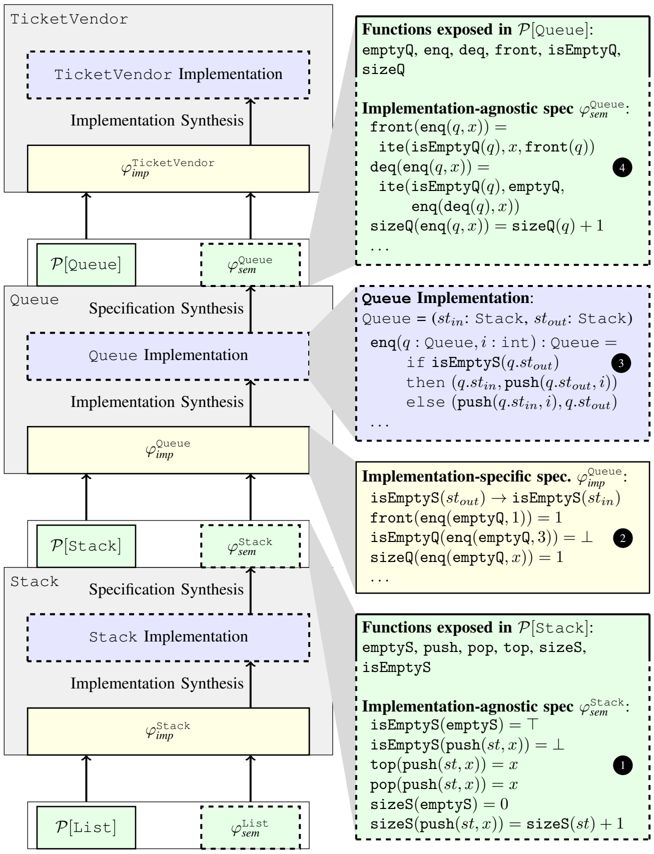
- The decomposition of the problem into `TicketVendor`, `Queue`, and `Stack` modules (gray boxes in Fig. 1).
- Which operations are to be exposed by each module, denoted by $\mathcal{P}[module]$—e.g., in Fig. 1, the `Queue` module exposes $\mathcal{P}[\texttt{Queue}]$, which contains `enq` and `deq` 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 `Stack`s.
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 $\mathcal{P}[\texttt{Stack}]$ and its (potentially synthesized) implementation-agnostic specification $\varphi_{sem}^{\texttt{Stack}}$—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

---

[1] 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.



Functions exposed in $\mathcal{P}$[Queue]:
emptyQ, enq, deq, front, isEmptyQ, sizeQ

Implementation-agnostic spec $\varphi_{sem}^{\texttt{Queue}}$: ④
$\texttt{front}(\texttt{enq}(q,x)) = \texttt{ite}(\texttt{isEmptyQ}(q), x, \texttt{front}(q))$
$\texttt{deq}(\texttt{enq}(q,x)) = \texttt{ite}(\texttt{isEmptyQ}(q), \texttt{emptyQ}, \texttt{enq}(\texttt{deq}(q), x))$
$\texttt{sizeQ}(\texttt{enq}(q,x)) = \texttt{sizeQ}(q) + 1$
...

Queue Implementation: ③
Queue = (st_{in}: Stack, st_{out}: Stack)
$\texttt{enq}(q : \texttt{Queue}, i : \texttt{int}) : \texttt{Queue} =$
if $\texttt{isEmptyS}(q.st_{out})$
then $(q.st_{in}, \texttt{push}(q.st_{out}, i))$
else $(\texttt{push}(q.st_{in}, i), q.st_{out})$
...

Implementation-specific spec. $\varphi_{imp}^{\texttt{Queue}}$: ②
$\texttt{isEmptyS}(st_{out}) \rightarrow \texttt{isEmptyS}(st_{in})$
$\texttt{front}(\texttt{enq}(\texttt{emptyQ}, 1)) = 1$
$\texttt{isEmptyQ}(\texttt{enq}(\texttt{emptyQ}, 3)) = \bot$
$\texttt{sizeQ}(\texttt{enq}(\texttt{emptyQ}, x)) = 1$
...

Functions exposed in $\mathcal{P}$[Stack]:
emptyS, push, pop, top, sizeS, isEmptyS

Implementation-agnostic spec $\varphi_{sem}^{\texttt{Stack}}$: ①
$\texttt{isEmptyS}(\texttt{emptyS}) = \top$
$\texttt{isEmptyS}(\texttt{push}(st, x)) = \bot$
$\texttt{top}(\texttt{push}(st, x)) = x$
$\texttt{pop}(\texttt{push}(st, x)) = x$
$\texttt{sizeS}(\texttt{emptyS}) = 0$
$\texttt{sizeS}(\texttt{push}(st, x)) = \texttt{sizeS}(st) + 1$

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 $\varphi_{imp}^{\texttt{TicketVendor}}$ 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 ($\varphi_{imp}^{\texttt{Queue}}$), shown in Fig. 1, contains, among others, constraints that state that (*i*) if the first stack st_{in} is empty, so is the second stack st_{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 ($\varphi_{imp}^{\texttt{Stack}}$) 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 $\varphi_{imp}^{\texttt{TicketVendor}}$ (and uses `Queue`); the `Queue` module implementation, which satisfies $\varphi_{imp}^{\texttt{Queue}}$ (and uses `Stack`); and the `Stack` module implementation, which satisfies $\varphi_{imp}^{\texttt{Stack}}$ (and uses lists). However, to synthesize the `TicketVendor` module implementation, we need an *implementation-agnostic specification* of `Queue`, denoted by $\varphi_{sem}^{\texttt{Queue}}$. The same can be said for the `Queue` module implementation, for which we need an implementation-agnostic specification of `Stack`, denoted by $\varphi_{sem}^{\texttt{Stack}}$.[2]

The user could write $\varphi_{sem}^{\texttt{Queue}}$ and $\varphi_{sem}^{\texttt{Stack}}$ 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 φ_{sem}^{M}, 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 $\mathcal{P}$[`List`] programs—that satisfies the implementation-specific specification $\varphi_{imp}^{\texttt{Stack}}$. (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 $\varphi_{sem}^{\texttt{Stack}}$, 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 $\varphi_{sem}^{\texttt{Stack}}$ is the conjunction of the equalities shown at ❶ in Fig. 1.

Using $\varphi_{sem}^{\texttt{Stack}}$ (❶), together with the implementation-specific specification $\varphi_{imp}^{\texttt{Queue}}$ (❷), we now synthesize the `Queue` module implementation (❸)—a set of $\mathcal{P}$[`Stack`] programs—and the implementation-agnostic specification $\varphi_{sem}^{\texttt{Queue}}$ (❹) via the same two-step process.

Finally, using $\varphi_{sem}^{\texttt{Queue}}$ and the implementation-specific specification $\varphi_{imp}^{\texttt{TicketVendor}}$, we synthesize the `TicketVendor` module implementation. (If needed by a further client, we would then synthesize the implementation-agnostic specification $\varphi_{sem}^{\texttt{TicketVendor}}$.) 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 $\varphi_{sem}^{\texttt{Stack}}$ and $\varphi_{sem}^{\texttt{Queue}}$.

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

---

[2] Technically, `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 $\varphi_{sem}^{\texttt{List}}$. In our evaluation, we synthesize $\varphi_{sem}^{\texttt{List}}$ automatically.

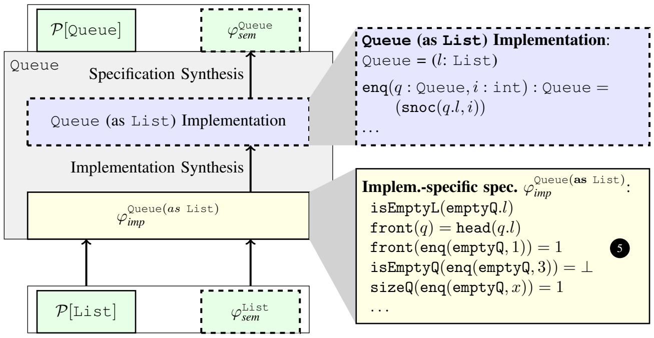


Fig. 2. Alternative implementation of the `Queue` module using list primitives instead of two stacks. $\mathcal{P}[\texttt{Queue}]$ and $\varphi_{sem}^{\texttt{Queue}}$ are the same as in Fig. 1.

symbols in $\mathcal{P}[\texttt{Queue}]$ and its implementation-agnostic specification $\varphi_{sem}^{\texttt{Queue}}$, 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 φ_{sem}^{MI} of the semantics of the module interface. Both MI and φ_{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.[3] 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 $\mathcal{P}$ (e.g., C with its core libraries), and use $\mathcal{P}[MI]$ to denote $\mathcal{P}$ 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 $\mathcal{P}[MI_{i-1} \cup PF_i \cup MI_i]$—i.e., f only uses operations in $\mathcal{P}$, 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).

A. Implementation-specific Specifications

When synthesizing specific implementations of the functions MI_i at layer i, the specifications are allowed to use symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$—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 $\varphi_{imp}^{MI_i}$ that only uses symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$.

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:

$$\texttt{isEmptyQ}(q) \iff \texttt{isEmptyS}(q.st_{in}) \land \texttt{isEmptyS}(q.st_{out}).$$

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

$$\texttt{isEmptyQ}(q) \iff \texttt{isEmptyL}(q.l).$$

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

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 $i+1$, to abide to the principle of information hiding, one cannot provide specifications that involve function symbols in $\mathcal{P}[MI_{i-1} \cup MI_i]$, but only those in $\mathcal{P}[MI_i]$.

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

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 $\varphi_{sem}^{\texttt{Queue}}$ for `Queue`:

$$\begin{array}{ll} \texttt{isEmptyQ}(\texttt{emptyQ}) = \top & \texttt{isEmptyQ}(\texttt{enq}(q, x)) = \bot \\ \texttt{sizeQ}(\texttt{emptyQ}) = 0 & \texttt{sizeQ}(\texttt{enq}(q, x)) = \texttt{sizeQ}(q) + 1 \\ \multicolumn{2}{l}{\texttt{front}(\texttt{enq}(q, x)) = \texttt{ite}(\texttt{isEmptyQ}(q), x, \texttt{front}(q))} \\ \multicolumn{2}{l}{\texttt{deq}(\texttt{enq}(q, x)) = \texttt{ite}(\texttt{isEmptyQ}(q), q, \texttt{deq}(\texttt{enq}(q), x))} \end{array} \tag{1}$$

Note that Eq. (1) serves as $\varphi_{sem}^{\texttt{Queue}}$ 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.

---

[3] In 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. 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 $\varphi_{sem}^{\texttt{Stack}}$ of `Stack`, *and* an implementation-specific specification $\varphi_{imp}^{\texttt{Queue}}$ that is allowed to talk about how the two `Stacks` used to implement a `Queue` are manipulated (e.g., $\texttt{isEmptyS}(st_{out}) \rightarrow \texttt{isEmptyS}(st_{in})$).

Definition 4 (Implementation synthesis): For module interface MI_i, the **implementation-synthesis** problem is a triple $(S_i, \varphi_{sem}^{MI_{i-1}}, \varphi_{imp}^{MI_i})$, where

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

A solution to the implementation-synthesis problem is an implementation of MI_i in S_i that satisfies $\varphi_{imp}^{MI_i}$.

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 $\varphi_{sem}^{MI_{i-1}}$ can talk about functions in MI_{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 $\texttt{isEmptyS}(\texttt{push}(st, x)) = \bot$.

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), $\varphi_{sem}^{MI_{i-1}}$ as a set of rewrite rules over the functions in MI_{i-1}, and $\varphi_{imp}^{MI_i}$ 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 $\varphi_{sem}^{\texttt{Queue}}$ of `Queue` is synthesized from the `Queue` implementation. (The same $\varphi_{sem}^{\texttt{Queue}}$, or one equivalent to it, is synthesized from the alternative `Queue` implementation of Fig. 2.)

Definition 5 (Specification synthesis): For module interface MI_i, a **specification-synthesis problem** is a pair (F_i, Φ_i) where

- F_i is a set of programs, written in $\mathcal{P}[MI_{i-1} \cup MI_i]$, that is a concrete implementation of MI_i.
- Φ_i is the set of possible properties we can use for $\varphi_{sem}^{MI_i}$ (every property in Φ_i uses only symbols in $\mathcal{P}[MI_i]$). (Typically, Φ_i is given as a regular-tree grammar for a fragment of logic in which terms can only use symbols in $\mathcal{P}[MI_i]$.)

A solution to the specification-synthesis problem is a set of properties $\varphi_{sem}^{MI_i} \subseteq \Phi_i$ such that for every $\alpha \in \varphi_{sem}^{MI_i}$:

Soundness: The implementation F_i satisfies α.

Precision: There is no property $\alpha' \in \Phi_i$ that implies α and such that the implementation F_i satisfies α'.

In general, there might not be just one answer to this synthesis problem because there could be multiple ways to build the set of properties $\varphi_{sem}^{MI_i}$. Furthermore, it can be the case that there are infinitely many properties in Φ_i that are sound, precise, and mutually incomparable. While in this paper we do not worry about these details, the tool we use in our evaluation SPYRO is always guaranteed to find a maximal set of properties in Φ_i whenever such a set is finite (SPYRO uses a regular-tree grammar to describe the set of possible properties Φ_i, but requires such a set to be finite.) In practice, even when the set is infinite, one can build tools that find a "good" set of properties and stop without trying to find an exhaustive set.

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 MI_1, MI_2, ..., MI_n (interleaved with the synthesis of $\varphi_{sem}^{MI_1}$, $\varphi_{sem}^{MI_2}$, ..., $\varphi_{sem}^{MI_n}$). In contrast, when the user is willing to write the implementation-agnostic specifications manually (in addition to the implementation-specific specifications $\{\varphi_{imp}^{MI_i}\}$), then the module implementations for MI_1, MI_2, ..., MI_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)

```
void snoc(list l, int val, ref list ret_list) {
   boolean is_empty_ret;

   ret_list = new list();
   is_empty(l, is_empty_ret);
   if (is_empty_ret) {
      ret_list.hd = val;
      nil(ret.tl);
   } else {
      ret_list.hd = l.hd;
      snoc(l.tl, val, ret.tl);
   }
}
```

Fig. 3. Implementation of **snoc** supplied to SPYRO. Returning a value from a function is done by storing the value into a reference parameter of the function.

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 *pattern* ⇒ *result*. For instance, one of the rewrite rules in the specification of a stack could be $\texttt{pop}(\texttt{push}(st, k)) \Rightarrow st$. To prevent infinite rewrite loops, a set of rewrite rules provided to JLIBSKETCH must not form a cycle. For instance, the rule $a+b \Rightarrow b+a$ is not allowed. The synthesis problem that JLIBSKETCH addresses is to find a program that is correct for any program input, for any library implementation that satisfies the algebraic specifications.

SPYRO addresses the problem of synthesizing specifications automatically, given an implementation. SPYRO takes as input (*i*) a set of function definitions Σ, and (*ii*) a domain-specific language $\mathcal{L}$—in the form of a grammar—in which the extracted properties are to be expressed. Properties that are expressible in $\mathcal{L}$ are called *$\mathcal{L}$-properties*. SPYRO outputs a set of $\mathcal{L}$-properties $\{\varphi_i\}$ that describe the behavior of Σ. Moreover, each of the φ_i is a *best* $\mathcal{L}$-property for Σ: there is no other $\mathcal{L}$-property for Σ that is strictly more precise than φ_i. Furthermore, the set $\{\varphi_i\}$ is *exhaustive*: no more $\mathcal{L}$-properties can be added to it to make the conjunction $\bigwedge_i \varphi_i$ more precise. SPYRO uses SKETCH as the underlying program synthesizer—i.e., it generates a number of synthesis problems in the form of SKETCH files and uses SKETCH to solve such problems.

Although SPYRO is built on top of SKETCH (instead of JLIBSKETCH), in our case study we manually implemented the term-rewriting approach used by the JLIBSKETCH solver in the SKETCH files used by SPYRO to synthesize implementation-agnostic specifications that only depend on algebraic specifications of lower layers. That is, we replace every function call f appearing in a SKETCH file with a function $normalize(f)$, where $normalize$ is a procedure that applies the rewrite rules from the algebraic specification.

MOSSKIT inherits the limitations of JLIBSKETCH and 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.

```
1  var {
2     int v1;
3     int v2;
4     list l;
5     list cons_out;
6     list snoc_out;
7  }
8  relation {
9     cons(v1, l, cons_out);
10    snoc(cons_out, v2, snoc_out);
11 }
12 generator {
13    boolean AP -> !GUARD || RHS;
14    boolean GUARD -> true
15                   | is_empty(l) | !is_empty(l);
16    boolean RHS -> equal_list(snoc_out, L);
17    int I -> v1 | v2;
18    list L -> l | nil()
19               | snoc(l, I) | cons(I, L);
20 }
```

Fig. 4. Grammar for the domain-specific language in which SPYRO is to express an extracted `List` property. The `relation` definition in lines 8-11 specifies that the variables `snoc_out` `l`, `v1` and `v2` are related by $\texttt{snoc\_out} = \textbf{snoc}(\textbf{cons}(\texttt{l}, \texttt{v1}), \texttt{v2})$. From the grammar (“`generator`”) in lines 12-20, SPYRO synthesizes best implementation-agnostic properties of form $\texttt{GUARD} \rightarrow \texttt{snoc\_out} = L$ (implicitly conjoined with $\texttt{snoc\_out} = \textbf{snoc}(\textbf{cons}(\texttt{v1}, \texttt{l}), \texttt{v2})$). In this case, the only expression for `GUARD` that succeeds is $\top$, and the property synthesized is $\texttt{snoc\_out} = \textbf{cons}(\texttt{v1}, \textbf{snoc}(\texttt{l}, \texttt{v2}))$ (with the additional implicit conjunct $\texttt{snoc\_out} = \textbf{snoc}(\textbf{cons}(\texttt{v1}, \texttt{l}), \texttt{v2})$).

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

```
 1 public void enq(int x) {
 2    Stack st_in = this.st_in;
 3    Stack st_out = this.st_out;
 4
 5    assume !st_out.isEmpty() || st_in.isEmpty();
 6
 7    if (genGuard(st_in, st_out)) {
 8       st_in = genStack2(st_in, st_out, x);
 9       st_out = genStack2(st_in, st_out, x);
10    } else {
11       st_in = genStack2(st_in, st_out, x);
12       st_out = genStack2(st_in, st_out, x);
13    }
14
15    assert !st_out.isEmpty() || st_in.isEmpty();
16
17    this.st_in = st_in;
18    this.st_out = st_out;
19 }
```

Fig. 5. JLIBSKETCH sketch of `enq`. Lines 5 and 15 assert the implementation-specific property $\texttt{isEmptyS}(st_{out}) \rightarrow \texttt{isEmptyS}(st_{in})$. JLIBSKETCH generates an expression to fill in each occurrence of the generators, `genStack2` and `genGuard`—the reader can think of each of these generators as being grammars from which JLIBSKETCH 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.

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 right-hand-side expression are one of the functions in the left-hand-side expression, functions used in the implementation of a function in the left-hand-side expression, or constants. Also, the outermost function symbol of the left-hand side can only be applied to a strictly smaller term.

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 $\varphi_{sem}^{\texttt{List}}$ synthesized by SPYRO is the following:

$$\begin{array}{ll}
\texttt{head}(\texttt{cons}(hd, tl)) = tl & \texttt{isEmptyL}(\texttt{nil}) = \top \\
\texttt{tail}(\texttt{cons}(hd, tl)) = hd & \texttt{isEmptyL}(\texttt{cons}(hd, tl)) = \bot \\
\texttt{sizeL}(\texttt{nil}) = 0 & \texttt{snoc}(\texttt{nil}, x) = \texttt{cons}(x, \texttt{nil}) \\
\multicolumn{2}{l}{\texttt{sizeL}(\texttt{cons}(hd, tl)) = \texttt{sizeL}(tl) + 1} \\
\multicolumn{2}{l}{\texttt{snoc}(\texttt{cons}(hd, tl), x) = \texttt{cons}(hd, \texttt{snoc}(tl, x))}
\end{array}$$

When considering the cumulative time taken to synthesize the algebraic specification of each module, SPYRO took 41 seconds for $\varphi_{sem}^{\texttt{List}}$ (longest-taking property 7 seconds), 34 seconds for $\varphi_{sem}^{\texttt{Stack}}$ (longest-taking property 7 seconds), and 44 seconds for $\varphi_{sem}^{\texttt{Queue}}$ (longest-taking property 13 seconds).

We used JLIBSKETCH 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-specific specification of the module to be synthesized. For example, the $\varphi_{imp}^{\texttt{Stack}}$ specification involved JLIBSKETCH code with 17 assertions, and the following examples are an excerpt from the $\varphi_{imp}^{\texttt{Stack}}$ specification (x, y, and z are universally quantified integers that are allowed to be in the range 0 to 10):

$$\begin{array}{rr}
\texttt{top}(\texttt{push}(\texttt{emptyS}, x)) = x & \texttt{top}(\texttt{push}(\texttt{push}(\texttt{emptyS}, x), y)) = y \\
\texttt{sizeS}(\texttt{emptyS}) = 0 & \texttt{sizeS}(\texttt{push}(\texttt{emptyS}, x)) = 1
\end{array}$$

Besides the assertions, we provided JLIBSKETCH with a fairly complete sketch of the structure of the implementation: we provided loops and branching structures, and only asked JLIBSKETCH to synthesize basic statements and expressions. For example, the sketch provided for the operation `enq` of module $\texttt{Queue} = (st_{in} : \texttt{Stack}, st_{out} : \texttt{Stack})$ is shown in Fig. 5. This sketch of `enq` of module `Queue` uses two `Stack`s: 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 $\varphi_{imp}^{\texttt{Queue}}$, shown as ② in Fig. 1.

Afterward, based on the implementation synthesized by JLIBSKETCH, SPYRO was able to solve each `Queue` specification-synthesis problem within 40 seconds, yielding the following implementation-agnostic specification $\varphi_{sem}^{\texttt{Queue}}$:

$$\begin{array}{ll}
\texttt{isEmptyS}(\texttt{emptyQ}) = \top & \texttt{isEmptyQ}(\texttt{enq}(q, i)) = \bot \\
\multicolumn{2}{l}{\texttt{sizeQ}(\texttt{emptyQ}) = 0} \\
\multicolumn{2}{l}{\texttt{sizeQ}(\texttt{enq}(q, i)) = \texttt{sizeQ}(q) + 1} \\
\multicolumn{2}{l}{\texttt{isEmptyQ}(q) \rightarrow \texttt{front}(\texttt{enq}(q, i)) = i} \\
\multicolumn{2}{l}{\neg\texttt{isEmptyQ}(q) \rightarrow \texttt{front}(\texttt{enq}(q, i)) = \texttt{front}(q)} \\
\multicolumn{2}{l}{\texttt{isEmptyQ}(q) \rightarrow \texttt{deq}(\texttt{enq}(q, i)) = q} \\
\multicolumn{2}{l}{\neg\texttt{isEmptyQ}(q) \rightarrow \texttt{deq}(\texttt{enq}(q, i)) = \texttt{enq}(\texttt{deq}(q), i)}
\end{array}$$

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 $\varphi_{imp}^{\texttt{TicketVendor}}$ consisted of JLIBSKETCH code with 24 assertions, and contains multiple examples, such as the following (again, x and y are universally quantified integers that are allowed to be in the range 0 to 10):

$$\begin{array}{l}
\texttt{numTicketsRem}(\texttt{prepSales}(2)) = 2 \\
\texttt{numWaiting}(\texttt{prepSales}(2)) = 0 \\
\texttt{numWaiting}(\texttt{resTicket}(\texttt{prepSales}(2), x)) = 1 \\
\texttt{issueTicket}(\texttt{resTicket}(\texttt{prepSales}(2), x)).owner = x
\end{array}$$

Again, we provided JLIBSKETCH with a fairly complete sketch of the program structure, and JLIBSKETCH was able to synthesize the implementations of all the `TicketVendor` functions within 10 seconds. For example, the function `prepSales` for $\texttt{TicketVendor} = (num_{ticket} : \texttt{int}, q_{waiting} : \texttt{Queue})$ was synthesized as $\texttt{prepSales}(n : \texttt{int}) := (n, \texttt{emptyQ})$.

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 $\varphi_{sem}^{\texttt{List}}$ took 3 seconds instead of the 2 seconds needed when the implementation of `List` was provided. Synthesizing `Queue` from the specification $\varphi_{sem}^{\texttt{Stack}}$ took 188 seconds instead of the 799 seconds needed when the concrete implementations of `Stack` and `List` were provided. Synthesizing `TicketVendor` from the specification $\varphi_{sem}^{\texttt{Queue}}$ took 7 seconds, but JLibSketch crashed when the concrete implementations of `Queue`, `Stack` and `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 implementations 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 `Queue` that uses `List` instead of two `Stack`s. The `List` holds the oldest element of the `Queue` at its head. The implementation-specific specification $\varphi_{imp}^{\texttt{Queue}\ (as\ \texttt{List})}$ consisted of JLibSketch code with 19 assertions, including examples similar to those shown at ⑤ in Fig. 2. We used JLibSketch to verify whether the specification $\varphi_{sem}^{\texttt{Queue}}$ still held true for the new implementation. Because it did (confirmation took <1 second), `TicketVendor` does not need to be changed to use the `Queue` (as `List`) 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 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 `ArrayList` and `HashMap` as underlying modules. The goal of their paper was to show that in 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 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 `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: `ArrayList`, `TreeSet`, `HashSet`, and `HashMap`. Each JLibSketch benchmark consisted of (i) an algebraic specification of the layer-0 module (written by hand), (ii) a 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., `HashSet` is implemented using an array). The mock is not needed by 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 JLibSketch experiments. From just the mock implementation of layer 0, we asked 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 Sketch-like specification of the layer-1 module, we asked MoSSKit to synthesize the implementation of layer 1. (The second step essentially replicated the algebraic-synthesis part of the 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, Spyro took 626 seconds for $\varphi_{sem}^{\texttt{ArrayList}}$, 54 seconds for $\varphi_{sem}^{\texttt{HashSet}}$, and 1,732 seconds for $\varphi_{sem}^{\texttt{HashMap}}$. Because mock implementations are simplified versions of actual implementations, the mock implementation of `TreeSet` is identical to the mock implementation of `HashSet`—i.e., they both represent sets as arrays. Furthermore, the two implementations have the same algebraic specifications—i.e., $\varphi_{sem}^{\texttt{HashSet}} = \varphi_{sem}^{\texttt{TreeSet}}$—which can thus be synthesized in the same amount of time.

Key finding: For all but two benchmarks, the $\mathcal{L}$-conjunctions synthesized by MoSSKit were equivalent to the algebraic properties manually written by Mariano et al. For the mock implementation of `HashMap` and `ArrayList` provided in JLibSketch, for specific grammars, MoSSKit synthesized empty $\mathcal{L}$-conjunctions (i.e., the predicate *true*) instead of the algebraic specifications provided by Mariano et al.—i.e., $k_1 = k_2 \Rightarrow \texttt{get}(\texttt{put}(m, k_1, v), k_2) = v$ and $i = j \Rightarrow \texttt{get}(\texttt{set}(l, i, v), j) = v$, for `HashMap` and `ArrayList`, respectively. Upon further inspection, we discovered that JLibSketch's mock implementation of `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 `HashMap`, we were able to synthesize the expected algebraic specification. However, when inspecting the implementation of `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 $(i = j) \wedge (0 \leq i) \wedge (i \leq \texttt{sizeL}(l)) \Rightarrow \texttt{get}(\texttt{set}(l, i, v), j) = v$. However, this modification revealed a bug in one of the implementations of `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 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) `HashMap` module from a (layer-0) algebraic specification of `ArrayList`. (The two benchmarks synthesized different implementations that handled collisions differently and we refer to the corresponding modules as `HashMap1` and `HashMap2`.) The third benchmark involved synthesizing the implementation of a (layer-2) `Kafka` from a (layer-1) algebraic specification of `HashMap`. Thus, we built two 3-layer benchmarks in which the goal was to synthesize `Kafka` using an implementation of `HashMap` that used an implementation of `ArrayList`. For us, each 3-layer benchmark involved four synthesis problems: (1) the algebraic specification $\varphi_{sem}^{\texttt{ArrayList}}$ 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 $\varphi_{sem}^{\texttt{ArrayList}}$ manually provided by Mariano et al., and were able to use to 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 $\varphi_{sem}^{\texttt{ArrayList}}$. However, the implementation of `HashMap1` synthesized by JLibSketch was incorrect because it depended on the original, erroneous specification $\varphi_{sem}^{\texttt{ArrayList}}$ 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 $\varphi_{sem}^{\texttt{ArrayList}}$ exposed by `ArrayList` still had a problem! In particular, $\varphi_{sem}^{\texttt{ArrayList}}$ was too weak to prove the algebraic specifications needed by `HashMap1` and `HashMap2`—i.e., $\varphi_{sem}^{\texttt{ArrayList}}$ did not characterize properties that were needed by `HashMap1` and `HashMap2` to satisfy the algebraic specification $\varphi_{sem}^{\texttt{HashMap}}$. We used Sketch itself to produce a violation of the algebraic specification $\varphi_{sem}^{\texttt{HashMap}}$ for `HashMap1` under the weaker assumption that `ArrayList` only satisfied the specification $\varphi_{sem}^{\texttt{ArrayList}}$, and used the violations generated by SKETCH to identify what properties we needed to add to strengthen $\varphi_{sem}^{\texttt{ArrayList}}$. In particular, $\mathtt{sizeL}(\mathtt{ensureCapacity}(l, n)) = \mathtt{sizeL}(l)$ and $\mathtt{get}(\mathtt{ensureCapacity}(l, n), i) = \mathtt{get}(l, i)$ were added to describe the behavior of `ensureCapacity`. We were then able to modify the grammar used to synthesize algebraic specifications for $\varphi_{sem}^{\texttt{ArrayList}}$ and synthesize the missing property. After obtaining $\varphi_{sem}^{\texttt{ArrayList}}$, we successfully synthesized the full algebraic specification for `HashMap2` (i.e., $\varphi_{sem}^{\texttt{HashMap}}$) 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 $\mathtt{get}(\mathtt{emptyMap}, k) = err$.

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 SPYRO. Besides SPYRO, 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 "bootstrapping" 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 directions 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 SPYRO [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

[1] R. Alur, P. Cerný, and A. Radhakrishna. Synthesis through unification. In D. Kroening and C. S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, volume 9207 of *Lecture Notes in Computer Science*, pages 163–179. Springer, 2015.

[2] J. Bessai, A. Dudenhefner, B. Düdder, M. Martens, and J. Rehof. Combinatory logic synthesizer. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change - 6th International Symposium, ISoLA 2014, Imperial, Corfu, Greece, October 8-11, 2014, Proceedings, Part I*, volume 8802 of *Lecture Notes in Computer Science*, pages 26–40. Springer, 2014.

[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. 2008.

[4] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[5] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 599–612, 2017.

[6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[7] J. Goguen, J. Thatcher, E. Wagner, and J. Wright. Abstract data-types as initial algebras and correctness of data representations. In *Proceedings Conference on Computer Graphics, Pattern Recognition and Data Structure*, May 1975.

[8] Z. Guo, D. Cao, D. Tjong, J. Yang, C. Schlesinger, and N. Polikarpova. Type-directed program synthesis for restful apis. In R. Jhala and I. Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 122–136. ACM, 2022.

[9] J. V. Guttag. *The Specification and Application to Programming of Abstract Data Types*. PhD thesis, Computer Systems Research Group, Univ. of Toronto, Toronto, Canada, Sept. 1975.

[10] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[11] J. Henkel, C. Reichenbach, and A. Diwan. Discovering documentation for Java container classes. *IEEE Trans. Software Eng.*, 33(8):526–543, 2007.

[12] R. Hood and R. Melville. Real-time queue operation in pure LISP. *Inf. Process. Lett.*, 13(2):50–54, 1981.

[13] B. H. Liskov and S. N. Zilles. Specification techniques for data abstractions. *IEEE Trans. Software Eng.*, 1(1):7–19, 1975.

[14] B. Mariano, J. Reese, S. Xu, T. Nguyen, X. Qiu, J. S. Foster, and A. Solar-Lezama. Program synthesis with algebraic library specifications. *Proc. ACM Program. Lang.*, 3(OOPSLA):132:1–132:25, 2019.

[15] K. Park, L. D'Antoni, and T. Reps. Synthesizing specifications. *CoRR*, abs/2301.11117, 2023.

[16] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(12):1053–1058, 1972.

[17] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, page 792–800. AAAI Press, 2015.

[18] K. Shi, J. Steinhardt, and P. Liang. FrAngel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, 2019.

[19] P. Simon. One man's ceiling is another man's floor, May 1973. T-700.050.850-1 BMI, ISWC, JASRAC.

[20] R. Singh, R. Singh, Z. Xu, R. Krosnick, and A. Solar-Lezama. Modular synthesis of sketches using models. In K. L. McMillan and X. Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer, 2014.

[21] J. L. Singleton, G. T. Leavens, H. Rajan, and D. R. Cok. Inferring concise specifications of APIs. *CoRR*, abs/1905.06847, 2019.

[22] A. Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

[23] J. M. Spitzen and B. Wegbreit. The verification and synthesis of data structures. *Acta Informatica*, 4:127–144, 1974.

[24] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021.