Enumerative Data Types with Constraints

Andrew T. Walter
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
walter.a@northeastern.edu

David Greve
Collins Aerospace
Cedar Rapids, IA, USA
david.greve@collins.com

Panagiotis Manolios
Khoury College of Computer Sciences
Northeastern University
Boston, MA, USA
pete@ccs.neu.edu

***Abstract*—Many verification and validation activities involve reasoning about constraints over complex, hierarchical data types. For example, distributed protocols are often defined using state machines that govern the behavior of processes communicating with messages which are hierarchical data types with state-dependent constraints and dependencies between component fields. Fuzzing, analyzing and evaluating implementations of such protocols requires solving complex queries that pose challenges to current SMT solvers. Generating fields that satisfy type constraints is one of the challenges and this can be tackled using enumerative data types: types that come with an enumerator, an efficiently computable function from natural numbers to elements of the type. Enumerative data types were introduced in ACL2s as a key component of counterexample generation, but they do not handle constraints such as dependencies between types. We extend enumerative data types with constraints and show how this extension enables applications such as hardware-in-the-loop fuzzing of complex distributed protocols.**

***Index Terms*—verification, data types, distributed systems, fuzzing, counterexample generation, ACL2s**

I. Introduction

The motivation for this paper stems from a project to analyze the IEEE 802.11 Wi-Fi protocol. Since the introduction of the first IEEE 802.11 standard in 1997 [1], the Wi-Fi family of protocols have become a key part of many user's ability to access the Internet. In 2019, Cisco predicted that over half of global Internet traffic will be transmitted over Wi-Fi and over 20% of global Internet traffic will be transmitted over a mobile network by 2022 [2]. Therefore, securing wireless networks and their underlying hardware is of critical importance. One method that researchers have used to demonstrate vulnerabilities in the Wi-Fi protocol is *fuzzing*, a form of testing in which generated data (possibly invalid) is input to a system, which is monitored for crashes, nonconforming responses, or other undesired behavior. Fuzzing has historically been successful in testing software systems, but bringing it into the realm of hardware raises several challenges.

Consider the general problem of validating the conformance of a given hardware device to a wireless protocol using hardware-in-the-loop fuzzing, where we have no internal knowledge of the device under test (DUT). In order to obtain good coverage of such protocols, we have to force the DUT into a variety of protocol states. Interesting protocols are nondeterministic, so we cannot easily precompute a set of messages to send; instead we must generate messages dynamically, in response to actual messages received from the DUT. Another complication is that protocols typically contain complex constraints on the format and contents of messages, making it infeasible to generate well-formed messages using standard fuzzing techniques. Finally, we note that such hardware devices are fast and associated protocols often involve short timeouts, on the order of hundreds of microseconds. Therefore, to effectively validate the protocol conformance of such devices, we must generate well-formed messages at high speeds.

The prevailing approach for message generation in scenarios like the above has been the development of custom software like Wifuzzit [3] and owfuzz [4]. Developing such software takes a significant amount of highly specialized engineering effort. A more general and powerful approach is to use formal methods to model the protocol under which the DUT is being tested and to then automatically generate protocol messages from that model, using formal methods tools. Unfortunately, current formal methods are not powerful enough to generate messages of the required complexity and at the required rate, as explained in detail later.

To address the above problem, we present *enumerative data types with constraints*, an idea that enables the fast generation of elements of hierarchical data types with constraints and inter-field dependencies. Our work is a natural extension of *enumerative data types* [5]: types that have *enumerators*, functions from natural numbers to elements of that type. We implemented the idea in the context of ACL2s [6], [7] and performed an evaluation by generating certain messages described in the 802.11 Wi-Fi protocol. Our evaluation shows that we are able to generate messages for a wide variety of sizes, something that neither SMT solvers nor pure enumerative data types can do. For the classes of messages that can also be generated by SMT or enumerative data types, our approach is at least two orders of magnitude faster.

Our contributions are as follows. (1) The idea of *enumerative data types with constraints*, which allows for the efficient generation of elements of dependent types with constraints and field interdependencies. (2) Extensions to the existing enumerative data type framework in ACL2s to support lists with length and ordering constraints, as well as improved support of numeric ranges. (3) The evaluation of our ideas with a case study on fuzzing Wi-Fi access points. All tools, models and artifacts developed for the case study, including sets of SMTLIB2-formatted constraints that may be useful

https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_25

This article is licensed under a Creative Commons Attribution 4.0 International License

```
(definec foo (x :int) :bool
  (!= x (expt 2 63)))
(property (x :int) (foo x))
$>...
We falsified the conjecture. Here are
counterexamples:
 --((X 9223372036854775808))
```

Fig. 1. A definition of a function and a property that ACL2s can find a counterexample to, but QuickCheck cannot in an equivalent Haskell formulation without the use of a custom generator.

for benchmarking SMT solvers will be publicly available [8]. (4) The idea of FM/hardware-in-the-loop for protocol conformance testing, where formal methods are used in the loop of a hardware-in-the-loop approach to protocol conformance testing.

The paper is organized as follows. Section II discusses related work in the areas of property testing, constraint-solver aided test data generation, and Wi-Fi fuzzing. Section III describes our extensions to enumerative data types and Section IV describes the idea of enumerative data types with constraints. A full, formal description is beyond the scope of the paper, due to the complexity of the data definition framework, but we have endeavored to present the ideas in a way that experts will be able to adapt them to other languages, type systems and tools. Section V discusses aspects of the implementation relevant for our Wi-Fi fuzzing case study, described in Section VI. Conclusions are presented in Section VIII.

II. Related Work

ACL2s (the ACL2 Sedan) [6], [7], is an extension of the ACL2 [9], [10] automated theorem prover that includes a powerful data definition framework (`defdata`) [5], a counterexample generation framework (`cgen`) for finding counterexamples to conjectures [11]–[13], a power termination analysis based on calling-context graphs [14] and ordinals [15]–[17] and IDE support in the form of an Eclipse plug-in.

QuickCheck [18] is a tool for performing property-based testing. It is emblematic of a family of tools that perform property-based testing of program without considering the formal semantics of those programs. Such tools are capable of finding many bugs, but there are many incorrect properties that they are highly unlikely to find counterexamples to without specific direction from the user. The `cgen` framework of ACL2s was inspired by QuickCheck and builds on it by combining random generation with theorem proving. Fig. 1 highlights an example of a function and property that ACL2s can find a counterexample to, but QuickCheck cannot in an equivalent Haskell formulation.

ACL2s is able to find a counterexample in the Fig. 1 example by making use of reasoning capabilities provided by ACL2. Note that `cgen` was able to produce this result without any property-specific configuration. `cgen` is successful because it is able to benefit from ACL2's process of transforming and splitting up the property being tested into smaller pieces. `cgen` also makes use of random testing during counterexample search. This random testing is deeply entwined with ACL2s' `defdata` data definition system for defining types [5]. `cgen` will be discussed in more detail in Section III.

Constraint Solvers and Test Data Generation: Outside of ACL2, many systems have been developed that allow the combination constraint solvers with models or specifications for the purpose of test data generation. The Alloy modeling language and its analyzer [19] constitute one such system: see Sullivan *et al.*'s framework for automated test generation in Alloy [20] as well as Abdul Khalek *et al.*'s use of Alloy to generate database management systems tests [21]. The Alloy analyzer's model-finding system differs substantially in approach from `cgen`—in particular, Alloy only supports bounded verification, meaning that it considers only a finite subset of all possible models, those with sizes in a user-provided bound, when verifying or searching for a counterexample to a property. Chamarthi *et al.* provide a detailed discussion of the differences between `cgen` and Alloy in [12], including that Alloy does not in general support recursive function definitions.

Other purpose-built systems include PLEDGE [22] and TAF [23]. Some of these systems attempt to generate test data that satisfies some coverage criterion of the given model; this is an interesting goal that is not described in this paper.

FuzzM [24] uses the JKind SMT-based model checker [25] to generate test data for fuzzing systems modeled in the Lustre programming language [26]. Depending on the complexity of the model provided, FuzzM may make queries to JKind that take a significant amount of time to solve. For this reason, FuzzM provides a generalization technique known as trapezoidal generalization [27] that can be used to generate many test data from a single datum produced by a query to JKind. Using trapezoidal generation with FuzzM can result in a data generation rate increase of several orders of magnitude.

Wi-Fi and Fuzzing: The Wi-Fi family of protocols is extensively used to provide local-area internet connections in a wide variety of settings including homes, businesses, and universities. Therefore, bugs and vulnerabilities in Wi-Fi protocols and implementations thereof can have a wide reach. For example, the 2017 KRACK attack [28] exposed a vulnerability in the 4-way handshake described by the 802.11 standard, affecting nearly every Wi-Fi device on the market at that time. The Wi-Fi protocols are based on the IEEE 802.11 standard [1], which describes the MAC (medium access control) and PHY (physical) layers of a network. We concern ourselves here with the MAC layer. The 802.11 standard describes the binary format of MAC frames, a generic overview of which is shown in Fig. 2.

Due to their prevalence, Wi-Fi protocols have previously been subjected to hardware-in-the-loop fuzz testing by several groups. In 2007, Laurent Butti and Julien Tinnés presented a hardware-in-the-loop approach [29] fuzzing Wi-Fi client drivers; this work resulted in the discovery of multiple bugs. Butti's 2007 system did not model the 802.11 MAC frame specification, and it instead focused on generating fuzzed

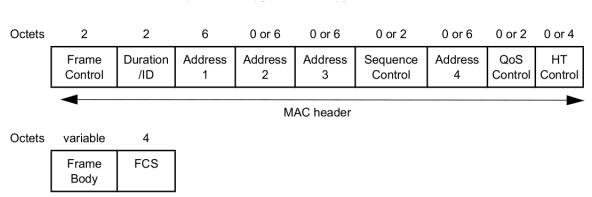

Figure 9-2—MAC frame format

Fig. 2. The binary layout of a generic 802.11 MAC frame. Figure taken from the IEEE 802.11-2020 standard [1].

frame elements and using the Scapy library [30] to generate packets with the appropriate structure that contain the fuzzed elements. More recently, Vanhoef *et al.* [31] described an approach for fuzzing access points' implementation of the 802.11 Wi-Fi handshake in which an abstract model of the Wi-Fi handshake is combined with test generation rules to produce test cases. These test cases consist of a sequence of abstract messages which are concretized into appropriate MAC frames when executed. This approach was able to find several vulnerabilities and quirks in the tested systems. In 2019, Garbelini *et al.* described their Greyhound system [32], which uses a model of the 802.11 protocol to generate frames that should drive the 802.11 client device into a particular protocol state before sending a fuzzed frame. Using a protocol model also allows Greyhound to analyze responses from the client device to determine if the client's responses comply to the 802.11 protocol. None of the aforementioned works regarding Wi-Fi fuzzing describe using theorem provers or constraint solvers to generate test data from protocol models. Based on our experience, we believe there would be a benefit to using constraint solvers in Wi-Fi protocol fuzzing, but the performance of existing approaches using constraint solvers is insufficient for use in the context. We will touch on this topic more in Section VI.

III. Enumerative Data Types

The idea of enumerative data types was introduced by Chamarthi *et al.* in the context of ACL2s and its `defdata` framework [5], a rich data definition framework that allows one to specify and reason about user-defined types. All `defdata` types have predicative characterizations in the form of *recognizers*, functions that recognize exactly the elements of the type, as well as enumerative characterizations in the form of *enumerators*, functions that, given a natural number, return an element of the data type. Enumerators in ACL2s are efficient, in part because they do not involve any theorem proving. In this section, we provide a short overview of `defdata` and present extensions to `defdata` that were added to support our application. These extensions are publicly available and formally verified using ACL2s.

The introduction of enumerative data types was partially motivated by counterexample generation and satisfiablity solving. ACL2s automatically generates counterexamples to function definitions and conjectures using a synergistic combination of theorem proving and enumerative data types. Theorem proving is used to decompose and simplify conjectures, at which point counterexample generation algorithms use type inference and enumerators to randomly generate elements based on the types of the variables appearing in the conjecture. In fact, counterexample generation in ACL2s uses enumerators and theorem proving in a recursive fashion, *e.g.*, after assigning a value to a variable, theorem proving is used to propagate consequences of the assignment, which may lead to further decompositions and simplifications as well as stronger type inferences, which are then exploited in further rounds of enumeration and theorem proving [11]–[13]. Satisfiability solving of ACL2s queries is performed similarly. This will be discussed in more detail in Section IV.

The `defdata` framework includes a large collection of built-in types. These types include basic types such as atoms, symbols, characters, strings, numbers and Booleans. Subtypes are supported and used extensively. Examples of subtypes include standard, non-special characters, keywords, symbols corresponding to variable names, and numeric types such as rationals, complex rationals, non-zero rationals, positive rationals, negative rationals, non-positive rationals, non-negative rationals, ratios (rationals that are not integers), positive ratios, negative ratios, integers, non-zero integers, natural numbers, positive integers, negative integers, non-positive integers, odd integers, even integers and zero. List and association list (alist) types, as well as non-empty versions, are also supported and are included for built-in types. There is also a universal type that includes all other types.

The `defdata` framework allows one to easily define new types by providing support for singleton types, enumeration types and range types (numeric ranges), as well as types built out of existing types, such as product types, union types, alias types, record types, list types, alist types, recursive types, mutually recursive types and map types (finite partial functions). The framework also allows one to define custom types, *e.g.*, to define the primes as a type, a user only needs to define a recognizer and an enumerator and then register the type. Custom types can then be used as if they were built-in to construct new types.

Polymorphic functions are also supported by `defdata`, *e.g.*, the form

```
(sig nth (nat (listof :a)) =>:a
  :satisfies (< x1 (len x2)))
```

states that `nth` is a function that given a natural number and a list of some type `:a` returns a list of type `:a`, as long as the first argument (`x1`) is less than the length of the list (`x2`).

The `defdata` framework automatically generates theorems in the form of various rules that ACL2s can use to reason about types using techniques such as rewriting, forward chaining, type reasoning, linear and non-linear arithmetic, as well as various decision procedures; see [9] for an in-depth discussion of the types of rules supported by ACL2. The framework includes support for specifying and reasoning about subtypes, *e.g.*, it includes and generates subtype theorems for built-in

and user-defined types. It also generates auxiliary functions, such as constructors and destructors, as appropriate.

Finally, the `defdata` framework includes numerous advanced features, *e.g.*, it allows users to select different randomization schemes, to define custom enumerators and to switch between enumerators dynamically.

We extended `defdata` by adding two libraries. The first library, `deflist`, provides support for defining list types with certain length and ordering constraints. The second library, `defintrange`, provides improved support for numeric range types over integers. The libraries are formally verified using ACL2 and are publicly available.

The `deflist` library provides the `defdata-list`, `defdata-ordered-list`, and `defdata-list-rng` forms, which are used to define `defdata` lists whose length is between two natural numbers, ordered lists with length constraints and lists with irregular length constraints, respectively. Consider the following example, derived from our Wi-Fi application:

```
(defdata-list SR8 SRType 1 8)
```

This defines the type `SR8`, which corresponds to lists whose length is between 1 and 8 (inclusive) of elements of type `SRType`, where `SRType` is a previous defined type recognizing 39 numbers between 2 and 236 that correspond to certain supported rates, as specified by the Wi-Fi protocol. The above form defines a recognizer and an enumerator for such lists. A type corresponding to lists of `SRType` with no length constraints is generated, if it does not already exist. Various tables keeping track of data types are updated. Rules for reasoning about lists of this type are also generated, *e.g.*, forward-chaining, type-prescription, compound-recognizer and rewrite rules that characterize the type and relate it to other types are automatically generated. Rules for reasoning about polymorphic functions and for controlling how the theorem prover uses these rules are also generated. This form generates a collection of forms totaling 7,944 lines and consisting of 434K bytes, all of which is formally verified by the ACL2 theorem prover.

The `defdata-ordered-list` form provides a similar capability but also imposes the constraint that the list is ordered. Consider the following example, derived from our Wi-Fi application:

```
(defdata-ordered-list BO255 uint8 0 255)
```

This defines the type `BO255`, which corresponds to lists of bytes (`uint8`) whose length is between 0 and 255 and whose elements are in increasing order. This form generates all of the forms that `defdata-list` generates, as well as rules for reasoning about the sorted lists. Finally, the `defdata-list-rng` form is similar to the `defdata-list` form, but allows one to specify irregular length constraints. Consider the following example, derived from our Wi-Fi application:

```
(defdata-list-rng BTS uint8 (gen-skip 22 254 2))
```

This defines the type `BTS`, which corresponds to lists of bytes (`uint8`) whose length is contained in the list of numbers generated by the form `(gen-skip 22 254 2)`, which includes the numbers $22, 24, \ldots, 254$. This form generates all of the forms that `defdata-list` generates, specialized to the irregular lengths.

The enumerators generated by the `deflist` library work by selecting a length in the appropriate range and then generating that many elements of the element type. This can be done very efficiently. If there are ordering constraints, then the generated list is sorted, using a verified sorting library we developed that includes an efficient sorting algorithm and supports sorting and potentially removing duplicates in the output. If duplicates are not allowed by the type, then they are removed, but this can result in lists whose length is shorter than desired. We experimented with a version of the library that generated lists of the appropriate length and where each such list had the same probability of being selected (*i.e.*, a uniform distribution), but that turned out to be computationally expensive for long lists. Therefore, once we sort the list and remove duplicates, we add a pass where we add elements not already in the list until we reach the target length. This turns out to be almost as fast as the non-ordered case.

The second library, `defintrange`, provides `defintrange` and `defnatrange` forms, which improved support for numeric range types over integers and natural numbers. Consider the following example, derived from our Wi-Fi application:

```
(defnatrange uint48 (expt 2 48))
```

This defines the type `uint48` which corresponds to the natural numbers less than 2^{48}. As was the case with `deflist`, we generate enumerators and rules for reasoning about the type, subtypes and polymorphic functions.

IV. Enumerative Data Types with Constraints

Complex data types often include type dependencies between fields. For example, consider a stack type which contains a field corresponding to the length of the stack with the type invariant that the value of this field is equal to the length of the stack. Sometimes there are dependencies between types, *e.g.*, a function may require that it is provided with two arguments, both of which are ordered lists of equal length. In this section, we show how to extend enumerative data types to support such constraints. The idea is relatively simple, but very powerful. As we show in this paper, this extension enables applications such as hardware-in-the-loop and theorem-prover-in-the-loop fuzzing of distributed protocols.

As a simple motivational example, consider a record consisting of n fields, $f_1, \ldots, f_n$, each of which is a list whose length is between 1 and 10 (inclusive). Before our work, an enumerator for f_i would generate a list of length l, with $1 \leq l \leq 10$ with probability $\frac{1}{10}$. However, suppose that we had a constraint that the size of the record, defined as the sum of the lengths of the fields, is $10n$. The probability of that happening, using the `defdata`-generated enumerator, is $\frac{1}{10^n}$, which for large n is essentially 0. Or, suppose that we have a dependent type where the lengths of the fields are required to be equal. The probability of that happening is $\frac{1}{10^{n-1}}$, which is also essentially 0 for large n.

The idea of enumerative data types with constraints is that we allow users to define types with parameters. These parameters are associated with functions over the data types and we require that, given values for these parameters, efficient enumerators for the types can be defined. For example, consider a list type with a parameter corresponding to the length of the list; the associated function is just the length function. Given a particular length, it is easy to generate a list of that length by generating the required number of elements using the enumerator for the element type. The next idea is to allow users to define constraints over the parameters and associated functions of types. If these constraints are over a decidable fragment of logic, then enumeration winds up becoming a two-stage process by which we find satisfying assignments to the constraints, providing values for the parameters, which are then used by the corresponding enumerators. Consider the motivating example where we had fields $f_1, \ldots, f_n$ with parameters $p_1, \ldots, p_n$, corresponding to the field lengths. The constraint that the size of the record is $10n$ gets turned into a constraint that the sum of the lengths of the fields, is $10n$ and this can be given to an SMT/IMT solver [33]–[35]. This is a simple constraint, which in terms of the parameters is $p_1 + \cdots + p_n = 10n$, and which only has one solution, namely $p_i = 10$. With the appropriate values for the parameters, we can now call the enumerators for the fields of the record, which will generate lists of the appropriate length, with probability 1. In general, enumerators require solving a set of constraints and then calling enumerators of component types, which may also require solving a set of constraints, and so on, recursively. As an optimization, recursive constraints associated with an enumerator can be packaged into single queries during the enumerator generation process, thereby minimizing the number of constraint-solving queries required by enumerators.

In our Wi-Fi application, and more generally in other verification efforts, we want to determine the satisfiability of a set of ACL2s constraints which include not only various data types, but also other constraints arising from a variety of sources, including coverage criteria, responses to messages from the DUT, well-formedness constraints, protocol constraints and modeling constraints. Queries to the underlying solver consist of the maximal subsets of these ACL2s constraints that can be expressed in the theory supported by the solver. If such a query is unsatisfiable, so is the corresponding ACL2s query; if the query is satisfiable, then we have values for the data type parameters which can be used to efficiently (without constraint solving) generate satisfying assignments to the datatype variables. If there are any remaining constraints, they are handled by the ACL2s counterexample generation process.

As we show later, we can formalize complex protocol interactions using types. These types include fields that are ordered lists over certain numbers, that have variable length and optional fields and that include other complex dependencies. Finding satisfying assignments to such types is difficult for current SMT solvers, but easy when using enumerative data types with constraints because we use constraint solving only for the true dependencies; we then we use the enumerative characterization of `defdata` to generate assignments using computation alone (*i.e.*, no constraint solving).

```
(solver-init)
(z3-assert (x :bool y :int z (:seq (:bv 3)))
    (and x (>= y 5) (= (seq.len z) y)))
(check-sat)
$> ;; This is SAT, so we get a model:
((X T) (Y 5) (Z (0 0 0 0 0)))
```

Fig. 3. An example showing the use of our Common Lisp-Z3 interface.

V. Implementation

We implemented enumerative data types with constraints in ACL2s, which provides support for defining tools on top of ACL2s via "ACL2s systems programming" [36]. We used Z3 as the constraint solver, which required that we integrate Z3 with ACL2s. To this end, we developed a library allowing one to easily call Z3 from Common Lisp. In this section, we will describe both the Common Lisp-Z3 interface library, and how we interacted with ACL2s.

Common Lisp-Z3 Interfacing: We decided to implement a close integration of ACL2 and Z3, using the CFFI Common Lisp library [37] to directly load Z3 into an ACL2s process and interact with it using Z3's C API. Such a close integration brings several benefits, including a low overhead when interacting with Z3 and the ability to support Z3 features like incremental solving. We developed our own Common Lisp library that provides both a low-level interface with Z3's C API and a high-level interface that allows the user to add assertions to Z3 using a syntax similar to that of ACL2s' `property` macro. See Fig. 3 for an example showing the use of our library. Our interface supports a broad swathe of Z3's features, including many of its built-in functions and types, several kinds of user-generated types and incremental solving.

ACL2s Interfacing: Since our system is implemented using the ACL2s systems programming paradigm, we are able to write Common Lisp code that calls into ACL2s. Our system starts inside the ACL2 read-eval-print loop (REPL), where we load in the ACL2s model that we will pull enumerators from. We then are able to exit from the ACL2 REPL into the underlying Common Lisp REPL that our copy of ACL2 is built on top of, where we can load any Common Lisp code that we might want, including our Common Lisp-Z3 library. To evaluate a function inside of ACL2—for example, an enumerator for a `defdata` type—we first generate an S-expression corresponding to the function call, and then pass that S-expression to the appropriate function provided by Walter *et al.*'s `acl2s-interface` library [38].

For our application, after running Z3 and getting back a length for each element of the structure being generated, we need to then generate elements with those lengths. Since each variable-length element has a list type corresponding to the set of bodies that it may have, we can make use of a special kind of enumerator that ACL2s produces for list types. This enumerator takes two arguments: the number of elements to generate, and the random seed to use. To generate an element

of a list type with a particular length, we simply call the enumerator with the desired length and an appropriate random seed. We can then construct our structure from its constituent parts by performing an appropriate ACL2s call.

VI. Wi-Fi Model Case Study and Evaluation

We present an application of enumerative data types with constraints to hardware-in-the-loop 802.11 wireless router fuzzing. We focus on the problem of generating a particular kind of 802.11 MAC frame, the *probe request* frame, as this is already sufficiently complex to present the challenges in modeling and frame generation. We first describe some challenges that come with hardware-in-the-loop fuzzing before discussing the probe request frame in more depth. We then discuss two models of the probe request frame that we developed, the first using Lustre and the second using ACL2s. We highlight the key challenges that arose when developing the Lustre model, and how we were able to use ACL2s to surmount these challenges and produce a more concise model. We then describe a system that implements enumerative data types with constraints alongside the ACL2s model, and conclude with experiments showing that our enumerative data type approach is able to generate probe request frames at a significantly greater rate and for a wider range of frame sizes than either a pure constraint solving approach or a pure enumerative data type approach.

Hardware-in-the-loop Fuzzing for Protocol Conformance

Fuzzing a hardware system like a wireless router brings with it certain requirements on the fuzzer and fuzzing infrastructure. The device under test (DUT) needs to be monitored, an interface must be formed between the DUT and the fuzzer, and in the case of protocol fuzzing, the fuzzer may be required to adhere to timing constraints imposed by the DUT. The latter constraint means that the performance of a fuzzer may not just affect how long it may take to find a particular vulnerability, but it may entirely preclude a fuzzer from use if it cannot generate a fuzzed response to a message sent by the DUT quickly enough.

The systems described below are intended to be one part of a larger hardware-in-the-loop fuzzing system, an architecture of which can be seen in Fig. 4. Each approach that we describe contains two parts: a model describing the probe request frame, and a fuzzer that uses the model to generate descriptions of concrete 802.11 probe request frames given some additional constraints on the size of the frame.

The Probe Request Frame

When a wireless device aims to connect to a 802.11 Wi-Fi access point, it must first gather information on the capabilities of wireless access points that are within range. To do this, the wireless device first sends out a *probe request* message with some basic information on its capabilities. Any Wi-Fi access point that is within range and supports at least one of the capabilities advertised by the wireless device will then respond with a *probe response* message containing information about itself. The wireless device will then select an access point to connect to and continue exchanging messages. The details of this process are described in the IEEE 802.11 specification [1]. Here we concern ourselves with the MAC frame corresponding to the probe request message.

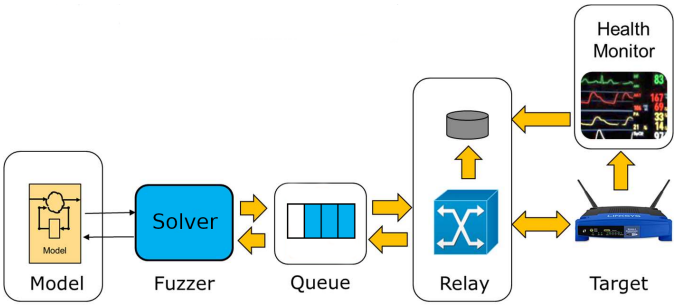

Fig. 4. An overview of a hardware-in-the-loop fuzzing architecture

The 802.11 specification states that every MAC frame consists of three parts: a header, a body, and a frame check sequence (FCS), which is a checksum for the previous two parts. We will not discuss the header and FCS parts, as the hardware-in-the-loop testing system can take care of setting the header and FCS as appropriate.

A probe request frame body consists of a variable-length sequence of elements, some of which are optional. Any elements that appear must appear in a specified order relative to each other. Elements typically contain a 1-byte "Element ID" field that has a constant value for all elements of a particular type, a 1-byte "Length" field that indicates the number of bytes remaining in the element after the end of the "Length" field, an optional 1-byte "Element ID Extension" field, and a variable-length set of element-specific fields. In this paper, we will consider the element-specific fields to all be concatenated into one "Body" field. The size of a probe request frame is the sum of the size of the MAC header (32 bytes) and the sizes of all elements appearing in the frame body. The 802.11 specification enumerates 33 element types for the probe request frame body, and the constraints on valid values for each element type vary widely. For example, the "DSSS Parameter Set" element's body is 1 byte long and should specify the "Current Channel" that the device is using; the set of valid values depends on the PHY implementation being used as well as well as some other settings. The "Request" element's body has a more complicated constraint: it is a variable-length list of bytes corresponding to "Element ID"s, and the bytes must be listed in increasing order. As we will see, such constraints are difficult to express in Lustre, and lead to a lengthy specification.

The Lustre Model

Our first model of the 802.11 probe request frame was developed using the Lustre programming language. When modeling the probe request frame body specification, we chose to abstract away some details of the specification in the interest of focusing on aspects of the specification that are interesting

```
type RequestElementType = struct {
  ElementID : byte     ;
  Len       : byte     ;
  Body      : byte[255] };
--Each element of the Body field is a byte.
node RequestElementTypeAssertions
  (e: RequestElementType) returns (r: bool);
let
r = ...
(0<=e.Len    ) and (e.Len    <=255) and
(0<=e.Body[0]) and (e.Body[0]<=255) and ...
(0<=e.Body[254]) and (e.Body[254]<=255);
tel
--The first Length elements of Body are
--sorted.
node RequestElementOrderedElementIDConstraint
  (e: RequestElementType) returns (r: bool);
let
r =
((e.Len<1) or (e.Body[0]<e.Body[1])) and ...
((e.Len<254) or (e.Body[253]<e.Body[254]));
```

Fig. 5. A code snippet highlighting how an element containing a variable-length sorted array of bytes is modeled in Lustre.

and representative. For example, we simply modeled the body of the DSSS parameter set element as a byte. In general, the Lustre model constrains the *shapes* of elements but not their body values, which we believe is reasonable considering the model is intended for use for fuzzing. That is, the Lustre model specifies probe request frame bodies that are of valid lengths and that have elements in the correct locations, but does not constrain the exact values that the body of each element may take to only those that are valid based on the 802.11 specification.

Lustre does not provide built-in support for bounded integer types, which means that specifying that a field is a byte is done by declaring that the field is an integer and that its value is between 0 and 255 inclusive. This becomes even more problematic when modeling variable-length arrays: to model an array of bytes of length between 0 and 255, the Lustre model specifies an array of length 255, specifies a variable representing the length of the array and adds a constraint for every element of the array stating that its value should be between 0 and 255 inclusive. This means that 255 array elements are always generated, and the system consuming values generated from the Lustre model simply omits any array elements that occur past the generated length value. Specifying the "Request" element is even more verbose, since in addition to the aforementioned constraints, 254 constraints are generated to specify that if the length of the array is greater than i, the element at index $i-1$ in the array is strictly less than the element at index i. See Fig. 5 for a snippet of the Lustre model that defines a frame element with a variable-length sorted array of bytes.

The Lustre model was used in conjunction with FuzzM to generate probe request frames. FuzzM was not able to generate assignments for certain frame sizes, as the SMT queries did not produce results even given a timeout of many minutes.

```
;; A natural number less than 256
(defnatrange uint8  (expt 2 8))
;; a list of uint8s with a length in [0,255)
(defdata-list byte255 uint8 0 255)
;; a byte255 that is also strictly ordered
(defdata-ordered-list byte255-increasing uint8
0 255)
;; Sanity check: should always be able to find
;; a byte255 that is not a byte255-increasing
(must-fail (property (x :byte255)
             (byte255-increasingp x)))
;; A type for the constant 10
(defdata exact10 10)
;; We model elements using records
(defdata RequestElementType
  (record (ElementID . exact10)
          (Body      . byte255-increasing)))
```

Fig. 6. A snippet of the ACL2s model showing how an element containing a variable-length sorted array of bytes is modeled. Also included are sanity checks that do not appear in the Lustre model.

The ACL2s Model

We developed an ACL2s model based on the Lustre model. The ACL2s model makes heavy use of `defdata`, which has a much more powerful notion of types than Lustre. The expressiveness of ACL2s allows us to more succinctly encode the constraints imposed by the 802.11 standard. `defdata` has built-in support for bounded integer types, making redundant many of the constraints that had to be stated explicitly in the Lustre model. We also used the extensions described in Section III to define list types with length bounds and ordering constraints. Fig. 6 shows all of the definitions necessary to model the "Request" element in ACL2s with our extensions.

Another benefit of developing our model in ACL2s is that we can include sanity checks inline with the model. ACL2s will evaluate the checks when the model is loaded during development, helping catch mistakes in the model specification that may otherwise go undetected. These checks can include validating that ACL2s can find a counterexample to a property (as seen in Fig. 6) but also may include proofs or code execution. If proofs are included, they may be used by ACL2s to prove or generate counterexamples to future conjectures. Even with sanity checks, the ACL2s version of the model has roughly a quarter of the lines of code present in the Lustre model.

Evaluation

We performed experiments to compare the performance of three approaches to probe request frame generation: enumerative data types using the ACL2s model and `cgen` (ACL2s-ET below), enumerative data types with constraints using the ACL2s model and an application-specific prototype of the approach described in Section IV (ACL2s-ETC below), and a pure constraint solving approach using a Z3-only version of the Lustre model and Z3 (Z3 below).

We measured the performance of each approach when queried for probe request frame bodies of various sizes, including sizes for which no probe request frame body exists.

Z3 and ACL2s were both configured to timeout after 20 seconds. ACL2s was set to use the `:uniform-random cgen` sampling method and was configured to terminate once it found a single counterexample rather than the default three; this brings its behavior more into line with Z3's. All other Z3 and ACL2s settings were left in their default state. We provide code for reproducing these experiments along with this paper.

Fig. 7 shows the number of query responses per minute for each approach across a range of probe request frame body sizes from 0 to 5000 bytes, sampled every 10 bytes. Five trials were performed for each frame size for all approaches. The number of query responses per minute for a particular approach and probe request frame body size was calculated by dividing the total number of queries made for that size that resulted in definitive responses (*e.g.* not timeouts) by the total amount of time in minutes spent on all queries for that size.

There are three regimes of frame size to discuss:

Small invalid probe request frame sizes (0-170 bytes): We expected all of the approaches to perform well in this regime. ACL2s-ETC consistently was able to determine UNSAT across this range of sizes, and the Z3-only approach performed well up to sizes of 150 bytes. ACL2s-ET was only able to determine sizes up to 30 bytes were UNSAT; all of the other queries in this regime resulted in timeouts. Note that Z3's performance begins to fall exponentially for frame sizes of 160 or greater.

Valid probe request frame sizes (180-2740 bytes): ACL2s-ETC is consistently able to generate frames at a rate greater than 1000 per minute, while ACL2s-ET is only able to generate frames for a subset of the frame sizes at a rate of at most 22 per minute and the Z3 approach is unable to generate any frames with a size greater than 300 bytes. The distribution of ACL2s-ET's response rate (approximately normally distributed around the average valid frame size of 1456 bytes) suggests that ACL2s is falling back on random generation of frame bodies; that is, generating a frame body by independently and randomly generating each element without consideration of the frame size constraint. The exponential drop in the Z3 approach's performance suggests that Z3's search space grows exponentially with frame size.

Large invalid probe request frame sizes (2750-5000 bytes): ACL2s-ETC is consistently able to quickly determine these sizes are UNSAT, while ACL2s-ET can do so slowly but consistently. The Z3 approach is always able to determine UNSAT, though it was only able to do so in all of the experimental trials in 100 of the 226 probe request frame sizes sampled between 2750 and 5000 bytes. This highlights inconsistency in Z3's ability to determine UNSAT for large frame sizes.

These results highlight the weaknesses of the Z3-only and ACL2s-ET approaches. The Z3 approach was able to quickly determine that small frame sizes are impossible and was consistently able to generate frames with sizes up to 210 bytes. However, the proportion of trials that resulted in SAT responses began to quickly drop after that point, and no SAT responses were received for trials with valid packet sizes of 290 bytes or greater. The Z3 approach's performance was highly variable for determining that larger frame sizes are impossible, and though it was inconsistent, it was always able to show UNSAT in at least one of the five trials performed. It is possible that an alternative encoding of the Z3 model (for example, one that does not make use of Z3's sequence types) would perform better, but our experience in using the Lustre model with FuzzM does not suggest a significant improvement in performance.

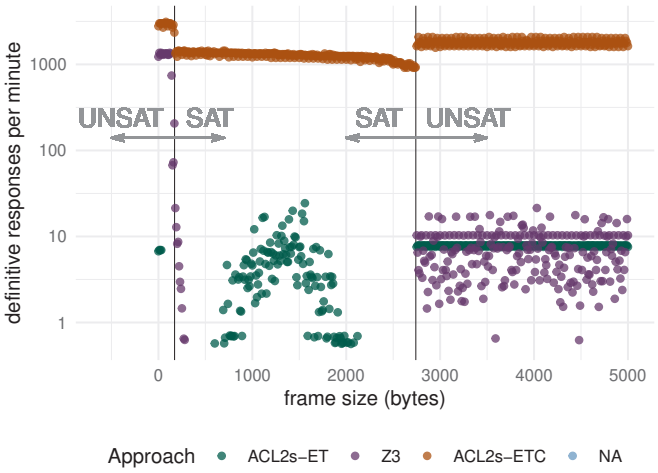

Fig. 7. The number of frames generated per minute using each of the three approaches when queried for frames with a given length. Only instances where the model returned a definitive response (*e.g.* not "unknown" or "timeout") are shown. The two vertical lines represent the minimum frame size and the maximum frame size; any responses outside of that range were all UNSAT, and any within that range were SAT.

The ACL2s-ET approach was consistently able to show that large frame sizes are impossible, and was able to generate frames for a wider range of frame sizes than the Z3 approach, though it struggled to generate large or small frames and to show that very small frame sizes are impossible. ACL2s is not using information from the frame size constraints to guide its counterexample generation in a meaningful way; `cgen` could be modified to improve its effectiveness here.

VII. Future Work

This paper introduces the idea of enumerative data types with constraints, or, equivalently, the idea of enumerative dependent types. We believe that this idea will be useful in many applications, *e.g.*, those requiring the analysis and verification of systems and models defined using dependent data types. Such applications include property-based testing, model-based development and distributed systems.

Below we provide a partial list of ideas for future work.

Formalizations and extensions: We plan on developing and formalizing the theory of enumerative data types with constraints for ACL2s and encourage others to develop similar formalizations for other dependent type systems and interactive theorem provers. We suspect that there are numerous interesting directions in which the basic approach can be extended to handle dependent logics of varying expressive power.

A specific extension of interest involves supporting relations of arbitrary arity, not just predicates. Conceptually this is straightforward: the relation can be turned into a predicate by combining all of the relation's arguments into a single value (a tuple or a record). Then, our approach allows us to represent and handle dependencies between the relation's arguments. A user can manually perform the conversion from relation to predicate, but ideally this could be done automatically.

ACL2 integration: We plan to provide first-class support for enumerative data types with constraints as part of the ACL2s `defdata` framework, so that ACL2s users can benefit from our work without needing to write custom code. Our proof-of-concept implementation used for this paper's evaluation uses ACL2s systems programming [36] techniques and is not integrated with ACL2s.

Optimizations: The ACL2s-ETC implementation evaluated in this work was not optimized, and we are confident that there are opportunities for both general and application-specific performance improvements in our method. One such optimization that we have experimented with in the context of stateful protocols is to perform offline (pre-enumeration) analyses of the protocol's state machine to identify how to efficiently explore interesting regions of the protocol's state space. This pre-analysis can significantly reduce the amount of work needed at enumeration time to generate appropriate responses to messages from the SUT. There are also interesting questions regarding coverage metrics and "fair" explorations that model analyses can help answer.

Model extraction: One limitation of our current work is that it requires models that are described using dependent types. An interesting question whether it is possible to provide automatic techniques that are able to take existing models and annotate them with the type information requires to use our work. This line of research can include the use of AI techniques such as Natural Language Processing (NLP) to automatically translate legacy prose descriptions of protocols into formal models that can be analyzed using our approach.

VIII. Conclusion

In this paper, we introduced the idea of enumerative data types with constraints. This allows us to use formal-methods-in-the-loop in the context of hardware-in-the-loop fuzzing for conformance testing of distributed protocols. We presented a case study where we modeled a portion of the IEEE 802.11 Wi-Fi specification and showed that we are able to generate messages for a wide variety of sizes, something that previous methods cannot do, thereby enabling the use of formal methods in new applications. Interesting directions for future work include adding such capabilities to other formal methods tools and using enumerative data types to analyze other distributed protocols.

Acknowledgments: This work was funded in part by the United States Department of the Navy, Office of Naval Research under contract N68335-17-C-0238. We thank Kristopher Cory and Grant Foudree for their support.

References

[1] "IEEE standard for information technology–telecommunications and information exchange between systems - local and metropolitan area networks–specific requirements - part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications," *IEEE Std 802.11-2020 (Revision of IEEE Std 802.11-2016)*, pp. 1–4379, 2021.

[2] J. Thomas Barnett, S. Jain, U. Andra, and T. Khurana. Cisco visual networking index (VNI) complete forecast update, 2017–2022. Cisco Systems, Inc. Accessed on May 21st, 2022. [Online]. Available: https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1213-business-services-ckn.pdf

[3] L. Butti. wifuzzit. [Online]. Available: https://github.com/0xd012/wifuzzit

[4] E7mer. owfuzz. [Online]. Available: https://github.com/alipay/Owfuzz

[5] H. R. Chamarthi, P. C. Dillinger, and P. Manolios, "Data definitions in the ACL2 sedan," in *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, F. Verbeek and J. Schmaltz, Eds., vol. 152, 2014, pp. 27–48.

[6] H. R. Chamarthi, P. Dillinger, P. Manolios, and D. Vroon, "The acl2 sedan theorem proving system," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2011, pp. 291–295.

[7] P. C. Dillinger, P. Manolios, D. Vroon, and J. S. Moore, "ACL2s: "the ACL2 sedan"," *Electronic Notes in Theoretical Computer Science*, vol. 174, no. 2, pp. 3–18, 2007, proceedings of the 7th Workshop on User Interfaces for Theorem Provers (UITP 2006). [Online]. Available: https://doi.org/10.1016/j.entcs.2006.09.018

[8] A. T. Walter, D. Greve, and P. Manolios. Enumerative data types with constraints supporting material. [Online]. Available: https://gitlab.com/acl2s/external-tool-support/enumerative-data-types-with-constraints-supporting-material

[9] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.

[10] M. Kaufmann and J. S. Moore, "ACL2 homepage," 2022. [Online]. Available: https://www.cs.utexas.edu/users/moore/acl2/

[11] H. R. Chamarthi, P. C. Dillinger, M. Kaufmann, and P. Manolios, "Integrating testing and interactive theorem proving," in *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, D. S. Hardin and J. Schmaltz, Eds., vol. 70, 2011, pp. 4–19.

[12] H. R. Chamarthi and P. Manolios, "Automated specification analysis using an interactive theorem prover," in *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11*, P. Bjesse and A. Slobodová, Eds. FMCAD Inc., 2011, pp. 46–53. [Online]. Available: http://dl.acm.org/citation.cfm?id=2157665

[13] H. R. Chamarthi, "Interactive non-theorem disproving," Ph.D. dissertation, Northeastern University, 2016.

[14] P. Manolios and D. Vroon, "Termination analysis with calling context graphs," in *Computer Aided Verification, 18th International Conference, CAV, Proceedings*, ser. LNCS, T. Ball and R. B. Jones, Eds., vol. 4144. Springer, 2006, pp. 401–414.

[15] ——, "Algorithms for ordinal arithmetic," in *19th International Conference on Automated Deduction – CADE-19*, ser. LNAI, F. Baader, Ed., vol. 2741. Springer–Verlag, July/August 2003, pp. 243–257.

[16] ——, "Integrating reasoning about ordinal arithmetic into ACL2," in *Formal Methods in Computer-Aided Design FMCAD*, ser. LNCS. Springer–Verlag, November 2004.

[17] ——, "Ordinal Arithmetic: Algorithms and Mechanization," *Journal of Automated Reasoning*, vol. 34, no. 4, pp. 387–423, 2005.

[18] K. Claessen and J. Hughes, "QuickCheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. ACM, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[19] D. Jackson, *Software Abstractions - Logic, Language, and Analysis*. MIT Press, 2006. [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=10928

[20] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for Alloy," in *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. IEEE Computer Society, 2017, pp. 264–275. [Online]. Available: https://doi.org/10.1109/ICST.2017.31

[21] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, "Query-aware test generation using a relational constraint solver," in *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*.

[22] G. Soltana, M. Sabetzadeh, and L. C. Briand, "Practical constraint solving for generating system test data," *ACM Transactions on Software Engineering and Methodology*, vol. 29, no. 2, apr 2020. [Online]. Available: https://doi.org/10.1145/3381032

[23] C. Robert, J. Guiochet, H. Waeselynck, and L. V. Sartori, "TAF: a tool for diverse and constrained test case generation," in *21st IEEE International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021. [Online]. Available: https://hal.laas.fr/hal-03435959

[24] R. Coppa, G. Foudree, and D. Greve, "FuzzM: A model-based approach to grey-box fuzzing," Rockwell Collins, Tech. Rep., 2018. [Online]. Available: http://loonwerks.com/publications/pdf/coppa2018techreport.pdf

[25] A. Gacek, J. Backes, M. Whalen, L. G. Wagner, and E. Ghassabani, "The JKind model checker," in *Computer Aided Verification - 30th International Conference, CAV 2018, Proceedings, Part II*, ser. LNCS, H. Chockler and G. Weissenbacher, Eds., vol. 10982. Springer, 2018, pp. 20–27. [Online]. Available: https://doi.org/10.1007/978-3-319-96142-2_3

[26] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[27] D. A. Greve and A. Gacek, "Trapezoidal generalization over linear constraints," in *Proceedings of the 15th International Workshop on the ACL2 Theorem Prover and Its Applications*, ser. EPTCS, S. Goel and M. Kaufmann, Eds., vol. 280, 2018, pp. 30–46. [Online]. Available: https://doi.org/10.4204/EPTCS.280.3

[28] M. Vanhoef and F. Piessens, "Key reinstallation attacks: Forcing nonce reuse in WPA2," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, p. 1313–1328. [Online]. Available: https://doi.org/10.1145/3133956.3134027

[29] L. Butti and J. Tinnés, "Discovering and exploiting 802.11 wireless driver vulnerabilities," *Journal in Computer Virology*, vol. 4, no. 1, pp. 25–37, 2008. [Online]. Available: https://doi.org/10.1007/s11416-007-0065-x

[30] P. Biondi. scapy. [Online]. Available: https://github.com/secdev/scapy

[31] M. Vanhoef, D. Schepers, and F. Piessens, "Discovering logical vulnerabilities in the Wi-Fi handshake using model-based testing," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2017*, R. Karri, O. Sinanoglu, A. Sadeghi, and X. Yi, Eds. ACM, 2017, pp. 360–371. [Online]. Available: https://doi.org/10.1145/3052973.3053008

[32] M. E. Garbelini, C. Wang, and S. Chattopadhyay, "Greyhound: Directed greybox Wi-Fi fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 2, pp. 817–834, 2022. [Online]. Available: https://doi.org/10.1109/TDSC.2020.3014624

[33] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Proceedings*, ser. LNCS, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340.

[34] P. Manolios and V. Papavasileiou, "ILP modulo theories," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 662–677.

[35] P. Manolios, J. Pais, and V. Papavasileiou, "The Inez mathematical programming modulo theories framework," in *International Conference on Computer Aided Verification*. Springer, 2015, pp. 53–69.

[36] A. T. Walter and P. Manolios, "ACL2s systems programming," in *Proceedings of the Seventeenth International Workshop on the ACL2 Theorem Prover and its Applications*, ser. EPTCS, 2022, to be published.

[37] J. Bielman and L. Oliveira. CFFI–the common foreign function interface. [Online]. Available: http://common-lisp.net/project/cffi

[38] P. Manolios and A. Walter. ACL2s interface. [Online]. Available: https://gitlab.com/acl2s/external-tool-support/interface