AFM'07: Second Workshop on Automated Formal Methods November 6, 2007 Atlanta, Georgia

John Rushby and Natarajan Shankar (Editors)

SRI International
Computer Science Laboratory
Menlo Park CA 94025 USA
{rushby | shankar}@csl.sri.com

Table of Contents

Preface

This volume contains the proceedings of the Second Workshop on Automated Formal Methods held on November 6, 2007, in Atlanta, Georgia, as part of the Automated Software Engineering (ASE) Conference. The first AFM workshop was held as part of the Federated Logic Conference in July 2006 in Seattle, Washington. The focus of the AFM workshop is on topics related to the SRI suite of formal methods tools including PVS, SAL, and Yices. We received 13 submissions of which 9 were accepted for presentation at the workshop.

In addition to the contributed papers, the conference included a presentation by John Rushby on *The Road Ahead for PVS, SAL, and Yices*, a session of short presentations of ongoing research, and a discussion of open source extensions and enhancements of these verification tools.

We thank the distinguished members of the program committee as well as the external referees for their thorough and thoughtful reviews of the submitted papers. The paper submission and reviewing process was managed through the Easychair conference management system. We also thank the organizers associated with ASE 2007, including the conference general chair Kurt Stirewalt, the program co-chairs Alexander Egyed and Bernd Fischer, the workshop chairs Neelam Gupta and George Spanoudakis, and the publicity chair Yunwen Ye. We also received help from the ACM Publications Coordinator Adrienne Griscti and our SRI colleagues Bruno Dutertre, Sam Owre, and Ashish Tiwari.

We hope the AFM workshop series will continue to serve as a forum for communication and cooperation between the developers and users of automated formal verification tools.

John Rushby
Natarajan Shankar
Menlo Park, California

Programme Chairs

John Rushby
Natarajan Shankar

Programme Committee

Tevfik Bultan
Marsha Chechik
Jin Song Dong
Jean-Christophe Filliatre
Bernd Finkbeiner
Marcelo Frias
Chris George
Mike Gordon
Constance Heitmeyer
Paul Jackson
Joseph Kiniry
Panagiotis Manolios
Paul Miner
David Monniaux
David Naumann
Paritosh Pandya
AndrÃs Pataricza
John Penix
Lee Pike
S Ramesh
Scott Stoller
Ofer Strichman
Neeraj Suri
Mark Utting
Michael Whalen
Brian Williams
Leonardo de Moura

External Reviewers

Mark-Oliver Stehr
Jun Sun
Yuzhang Feng

Preliminary Explorations in Specifying and Validating Entity-Relationship Models in PVS

Venkatesh Choppella
Indian Institute of Information Technology and Management – Kerala
Thiruvananthapuram, India

Arijit Sengupta
Wright State University
Dayton, OH, USA

Edward L. Robertson
Indiana University
Bloomington, IN, USA

Steven D. Johnson
Indiana University
Bloomington, IN, USA

ABSTRACT

Entity-Relationship (ER) diagrams are an established way of doing data modeling. In this paper, we report our experience with exploring the use of PVS to formally specify and reason with ER data models. Working with a text-book example, we rely on PVS's theory interpretation mechanism to verify the correctness of the mapping across various levels of abstraction. Entities and relationships are specified as user defined types, while constraints are expressed as axioms. We demonstrate how the correctness of the mapping from the abstract to a conceptual ER model and from the conceptual ER model to a schema model is formally established by using typechecking. The verification involves proving the type correctness conditions automatically generated by the PVS type checker. The proofs of most of the type correctness conditions are fairly small (four steps or less). This holds out promise for complete automatic formal verification of data models.

Keywords

Data modeling, entity-relationship diagrams, mapping, formal methods, PVS, type checking, data refinement.

1. INTRODUCTION

Data modeling is a fundamental prerequisite for the physical design and implementation of a database system. Data modelers analyze the user's requirements and build *data models*, which are conceptual representations of real world enterprises.

A data model consists of a set of type, function, relation and constraint definitions. This model is validated for *consistency* and then used as a reference for further design refinements and *implementation*. The model serves as a *specification* to which the database design, usually specified in the form of a set of schemata, must conform. The most popular conceptual modeling framework is the *entity-relationship (ER) model* [11]. An ER model consists of a collection of *entities*, and *attributes* of and *relationships* among those entities. In addition, the model specifies *constraints* between its various entity and relationship sets.

A good modeling framework should allow a designer to (*a*) *express* and *reason* about data models at a high level of abstraction in a semantically precise manner, (*b*) *validate* the correctness of models across various levels of abstraction, and (*c*) *explore* design alternatives within correctness boundaries. In this paper, we explore the first and second problems: how to specify models at varying levels of abstraction using a specification language and then validate mappings between abstraction levels. We plan to address the third issue in a future paper.

1.1 Expressivity and correctness of mapping in ER modeling

ER models have an underlying formal semantics based on the elementary theory of sets and relations. Data modelers, however, prefer to employ ER diagrams, which are annotated, undirected graphs. The vertices in these graphs are the objects of the model: attributes, entities, and relationships. These objects are connected by edges which related these objects. In addition, an ER model consists of constraints. A limited set of decorations on vertices and edges encode key attributes, and participation and cardinality constraints. The diagrammatic approach allows for easy construction and intuitive understanding of models. But such notation does not easily extend to encoding arbitrary constraints arising from complex business logic. As a consequence, it becomes difficult for designers to express, much less prove, the correctness of their conceptual design and its mapping to a relational implementation. Designers therefore employ natural language to informally express nontrivial constraints. Natural language complements diagrammatic notation, but it is often the source of inaccuracies and ambiguities in specifications.

1.2 Specifying data models in PVS

How well can we apply the principles and techniques from formal specification languages to the ER data modeling prob-

AFM'07, November 6, Atlanta, GA, USA.

lem? We address this question by exploring the use of the Prototype Verification and Specification (PVS) language [2] for doing ER modeling. We look at two specific subproblems: (*a*) how to specify and reason with a data model at the abstract, ER and schema level, and, (*b*) how to prove that the mapping from abstract to ER and ER to schema is correct. Our preliminary experience indicates that the generality of PVS's specification language, its rich type system and libraries allow us to *reason* with our models in ways not easily possible with ER diagrams. Furthermore, this reasoning forms the basis of verifying the correctness of mapping between an ER model and the relational schemata that represent its implementation.

The mapping across different layers of abstraction of the ER model is an instance of the more general problem of *data refinement*: abstract, uninterpreted types and objects over those types at higher levels of abstractions are provided interpretations at the lower level of abstractions. To be sure, the problem of specification and the step-wise refinement to implementation has a long history in formal methods, software engineering and data models as well. This includes formally specifying the "Norman's database" example in VDM, RAISE and COLD-K [16, 18, 20, 37, 39]. Additionally, languages like Z, B and Alloy also come with various levels of automated support for data refinement and formal validation [1, 4, 22].

PVS is a modern, general purpose specification language with support for higher-order logic, dependent types, and interactive theorem proving environment for verification and typechecking. It is therefore natural to explore how well ER modeling can be done in PVS.

All specifications, including requirements, conceptual data models and logical models are expressed as *theories* in PVS. A theory is a basic PVS module and consists of declarations that define types, functions, axioms and theorems. Model-specific data constraints are encoded as axioms. Hard-to-anticipate constraints governing interactions between the various elements of the model are generated automatically *type correctness conditions* (TCCs). The modeler interactively verifies these TCCs to ensure that the model is consistent. The specification's correctness depends on verifying the type correctness conditions. For the example discussed in this paper, the majority of TCCs have proofs that are quite small and elementary. In general, however, the TCCs can be hard or even impossible to prove. In the latter case, this means that the specification is erroneous. PVS typechecking is undecidable, and the modeler needs to interactively prove theorems to typecheck a specification. Reasoning also proceeds by the modeler declaring, and then interactively proving lemmas about the specification. The result is a more powerful notation that allows arbitrary constraints to be expressed precisely and unambiguously using a high degree of abstraction.

For specifying ER models, we rely on the use of modeling constructs like functions and their various kinds (injections, partial functions, etc.). For example, it seems more natural to model certain weak entities using functions rather than relationships. In going from abstract to ER to schema, we apply data refinement to map uninterpreted types first to nested record types and then to flat record types. PVS implements data refinement via the principle of *theory interpretations*, an idea from universal algebra and logic in which the axioms of one theory are interpreted as theorems by another [31]. We specify the example model as three separate theories, each capturing on level of abstraction: the first is an abstract model in which entity, attribute and relation types are parameters. The second is an ER model defined by instantiating the abstract theory with concrete record types for entities and relations. The third is a schema model in which entities and relations are implemented as flat records. We then verify the correctness for the two mappings: from abstract to ER, and ER to schema. This allows one to formally state and prove that a data model at one level of abstraction implements another, more abstract model.

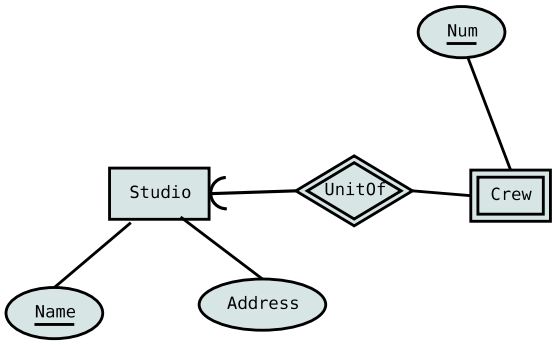


Figure 1: ER diagram slice of Ullman and Widom example [35, Chapter 2].

Outline of rest of the paper:

The rest of the paper consists of the following sections: Section 2 introduces an example model using an ER diagram and points to specific limitations of the diagram approach. Section 3 defines an abstract data model for the mini-example. Section 4 defines a record-based ER model for the mini-example. Section 5 presents a schema-based model of the mini-example. The next section, Section 6 addresses the issue of correctness of the mapping from the ER to the schema level. Section 7 discusses the results of the implementation: the sizes of the theories, and effort involved in proving type correctness conditions and user defined lemmas. Section 8 compares our work with existing approaches to data modeling in the literature. Section 9 discusses future work and Section 10 concludes the paper.

2. EXAMPLE ER MODEL

We pick the movies example data model from Ullman and Widom's introductory college text on databases to explore the approach of formally specifying and verifying a data model [35, Chapter 2]. Due to limited space, we focus on a self-contained slice of the example. The ER diagram of the slice is shown in Figure 1. The PVS specification of the complete example is discussed in an earlier technical report [12]. PVS source code for the complete example is available online [3].

The model consists of studio and crew entities. A crew is a unit of a studio. A studio has a name and address, whereas a crew has a number.

An ER model defines the *types* of attributes, entities and relationships. It also defines specific *sets* of entities and relationships *over* these types. The diagram does not distin-

guish between the types and the sets. When formalizing the model, however, we need to make the distinction explicit. Therefore, we use `Studio`, `Crew`, and `Unit_Of` for entity and relationship types, and `studios_set`, `crews_set`, and `unit_of_set` for entity and relationship sets, respectively.

2.1 Constraints

In addition to the entity and relationship types and sets, in the model shown in Figure 1, the following four constraints (of a total of twelve for the entire example) govern the model's semantics: key constraints (1) and (2), cardinality constraint (3), and referential integrity constraint (4).

1. The Name attribute is a key for `studios_set`.
2. The Num (number) attribute and `studios_set`, via the `unit_of_set`, together form a key for `crews_set`. (See cardinality constraint 3.)
3. `unit_of_set` is many-to-one from `crews_set` to `studios_set`.
4. For every element in `unit_of_set`, the constituent components belong to `studios_set` and `crews_set`.

In ER diagrams, key constraints are expressed by underlining key attribute names. `crews_set` is *weak*; its key is defined in terms of attributes from supporting relationships in which it participates. In ER diagrams, double borders identify weak entities and their supporting relationships. A many-to-one relationship has a round arrow on the edge at the "one" end the relation. Referential integrity means that the entity components of each element of a relationship set belong to their respective entity sets.

In the following sections, we build PVS theories for the example model, one for each level of abstraction: a parameterized abstract level, ER level, and the schema level. The type structure, entity set structure and the axioms used to specify the constraints vary for each level of abstraction.

3. ABSTRACT DATA MODEL

An abstract specification of a data model consists of the following kinds of objects:

- Attributes, entities, and relationship types, which are all uninterpreted. We call these *abstract entity types*.
- *Maps*, which connect abstract entity types. Some maps – those in correspondence with edges in the ER diagram, but oriented – are called *projectors*.
- *Abstract entity sets*, which are sets over the corresponding entity types.
- *Constraints*, which are specified as axioms over abstract entity sets, abstract entity types, and maps, and other functions.

The abstract specification is parameterized on the abstract entity types, maps, and the abstract entity sets.

Continuing the example, `movie_param_abstract` (Listings 3.1-3.7) is a specification of the abstract model of the example parameterized by abstract entity types (Listings 3.1), projectors (Listing 3.2), and abstract entity sets (Listing 3.3). The keyword TYPE+ posits that the types are non-trivial (i.e., nonempty). The abstract entity types and projectors together match the structure of the ER diagram in Figure 1.

Listing 3.1 (Abstract entity types).

```
movie_param_abstract[Name, Address, Num: TYPE+,
  Studio, Crew: TYPE+, UnitOf: TYPE+,
```

Listing 3.2 (Projectors).

```
studio_name:    [Studio -> Name],
studio_address: [Studio -> Address],
crew_num:       [Crew -> Num],
unit_of_crew:   [UnitOf -> Crew],
unit_of_studio: [UnitOf -> Studio],
```

Listing 3.3 (Abstract entity sets).

```
studios_set: set[Studio],
crews_set: set[Crew],
unit_of_set: set[UnitOf]]: THEORY
```

3.1 A theory for keys

In ER modeling, a *key* is an attribute or set of attributes that uniquely determine an element of an entity set. In our formalization of the abstract model, a key is identified not by an attribute name, but by a *key function*, which is often built using projectors that are injective. This is the natural way to model keys, since attributes of an entity are accessible using projector functions emanating from the entity.

Listing 3.4 (A theory for keys).

```
key[D:TYPE, S:set[D], R:TYPE, f:[D -> R]]: THEORY
  BEGIN
    ASSUMING
      restriction_is_injective: AXIOM
        injective?[(S), R]
          (restrict[D,(S),R](f))
    ENDASSUMING

    image_f_S: set[R] = image[D, R](f,S)
    I: TYPE = (image_f_S)
    h(s:(S)): I = f(s)

    h_is_bijective: LEMMA bijective?(h)
    getForKey: [I -> (S)] = inverse_alt(h)
    forKey(r: R):  lift[(S)] =
      IF (member(r,image_f_S))
        THEN up(getForKey(r)) ELSE bottom ENDIF
  END key
```

The theory for keys in Listing 3.4 defines the condition under which an abstract attribute entity type R is a key for uniquely identifying entities in a set S of elements of type D. The goal is to identify a key function that maps a key to a value in the entity set, if it exists. The function $f : D \to R$ is often a projector, retrieving an attribute in R from an entity in D. The elements of R qualify as keys provided the restriction of f to S is injective. The axiom in the theory captures this assumption. To see why this formulation implies the existence of a key function, let $I \subseteq R$ be the image of f on S. Since f restricted to S is injective, $h : S \to I$ defined as equal to f over S is a bijection. Therefore the function g from R to the lifted domain $S_\perp$ is a key function. g extends the bijective function $h^{-1} : I \to S$ to the domain R and range $S_\perp$. For an element $k \in R$, g maps k to $h^{-1}(k)$, if k is I, and to $\perp$ otherwise.

We now instantiate the `key` theory with different entity types and sets to obtain specific key constraints. The axiom in Listing 3.5, line 33 posits the injectivity of the restriction

Listing 3.5 (Key constraint on *studios_set*).

```
studio_name_injective_on_studios_set:
 AXIOM
injective?[(studios_set), Name](
 restrict[Studio, (studios_set),Name]
  (studio_name))

IMPORTING key[Studio, (studios_set),
  Name, studio_name]  AS studio_key

studio_for_name: [Name ->
 lift[(studios_set)]] =  studio_key.forKey
```

Listing 3.6 (Referential integrity of *unit_of_set*).

```
unit_of_ref_integrity: AXIOM
 FORALL (u: (unit_of_set)):
  member(unit_of_studio(u),studios_set)
  AND member(unit_of_crew(u),crews_set)
```

of the projection `studio_name` to `studios_set`. The axiom justifies the existence of a *key function* `studio_for_name` mapping names to the lifted domain of studios (line 42).

3.2 Referential integrity constraints

Referential integrity is specified in terms of projector functions and abstract entity sets. If $f : A \longrightarrow B$ is a projector from abstract entity type A to B, and a and b are, respectively, the entity sets of type A and B, then, referential integrity on the projector f emanating from the abstract entity type A of a is the property that $\forall x \in a.f(a) \in b$. Coming back to our example, the axiom on line 49 of Listing 3.6 is the referential integrity constraint on the projectors of `unit_of_set`.

3.3 Cardinality constraints

We consider the specification of the cardinality constraint on `unit_of_set` as an example.

In Listing 3.7, the image of the derived projector function `unit_of_crew_studio` on `unit_of_set` is used to define the binary relation `unit_of` (line 62). The cardinality constraint on `unit_of_set` boils down to declaring that the binary relation is a total function from `crews_set` to `studios_set`. This yields two projectors `crew_studio` and `crew_studio_num`.

3.4 Weak entities and foreign keys

Listing 3.8 shows how weak entities and foreign keys are specified at the abstract level. The injectivity of the derived projector `crew_studio_num` is used to axiomatize the key constraint on `crews_set` and yield the key function

Listing 3.7 (Cardinality constraint on *unit_of_set*).

```
unit_of_crew_studio(u:UnitOf):
 [Crew, Studio] =
   (unit_of_crew(u), unit_of_studio(u))

unit_of: set[[Crew,Studio]] =
 image(unit_of_crew_studio,unit_of_set)

function_unit_of:  AXIOM
 FORALL (cr: (crews_set)):
  exists1(LAMBDA(s: (studios_set)):
    unit_of(cr,s))
```

Listing 3.8 (Key constraint on *crews_set*).

```
crew_studio(cr: (crews_set)): (studios_set)
 = the(s:(studios_set) | unit_of(cr,s))

crew_studio_num(c:(crews_set)):
 [Studio, Num] =
  (crew_studio(c), crew_num(c))

crew_studio_num_injective_on_crews_set:
  AXIOM
   injective?[(crews_set), [Studio, Num]]
    (crew_studio_num)

% crew_for_studio_num  % key
```

Listing 4.1 (Attr. and Entity Types).

```
movie_rec: THEORY
  BEGIN
% Attribute Types
% ---------------
   NameEntity: TYPE+
   AddressEntity:  TYPE+
   NumEntity:  TYPE+

%  Entity and Relationship Types
%  -----------------------------
   StudioEntity: TYPE = [# name: NameEntity,
    address: AddressEntity #]
   studio_entity_name(s:StudioEntity)
    : NameEntity = s`name
   studio_entity_address(s:StudioEntity)
    : AddressEntity = s`address

   CrewEntity: TYPE =
    [# num: NumEntity, studio: StudioEntity #]
   crew_entity_num(c: CrewEntity)
    : NumEntity = c`num
   crew_entity_studio(c: CrewEntity)
    : StudioEntity = c`studio

   UnitOfEntity: TYPE =
    [# crew: CrewEntity, studio: StudioEntity #]
   unit_of_entity_crew(unit_of: UnitOfEntity)
    : CrewEntity  = unit_of`crew
   unit_of_entity_studio(unit_of: UnitOfEntity)
    : StudioEntity  = unit_of`studio
  END movie_rec
```

`crew_for_studio_num`. The entity set `crews_set` is weak; the projection function `crew_studio_num` involves `unit_of_set`, which is an abstract entity "foreign" to `crews_set`.

4. RECORD-BASED ER MODEL

At the abstract level discussed in the previous section, we do not distinguish between attribute, entities and relationship types. Nor is the internal structure of these types revealed. At the ER level, entity and relationship types are records. Attribute types, are, however, left uninterpreted because their structure has no role to play at this level. The record types may be nested, as in the case of relationship types. The record types for the example are defined in the theory `movie_rec` (Listing 4.1). Projectors now correspond to record selectors. The infix back quote operator in the PVS code indicates record selection.

4.1 Instantiating abstract to ER

The ER model is obtained from the parameterized abstract model by instantiating the abstract theory with suitable types, both concrete and abstract (uninterpreted). The theory `movie_er` (Listings 4.2–4.3) specifies the ER model

Listing 4.2 (Entity and Relationship sets).

```
movie_er: THEORY                                    14
 BEGIN                                              15
   IMPORTING props                                  16
   IMPORTING movie_rec                              17
                                                    18
    studios_entity_set:   set[StudioEntity]         19
    crews_entity_set:     set[CrewEntity]           20
    unit_of_entity_set:   set[UnitOfEntity]         21
```

Listing 4.3 (Instantiating *movie_param_abstract*).

```
   IMPORTING movie_param_abstract[                   23
    NameEntity, AddressEntity, NumEntity,           24
    StudioEntity, CrewEntity,  UnitOfEntity,        25
                                                    26
    studio_entity_name, studio_entity_address,      27
    crew_entity_num,                                28
    unit_of_entity_crew, unit_of_entity_studio,     29
                                                    30
    studios_entity_set, crews_entity_set,           31
    unit_of_entity_set]                             32
  END movie_er                                      33
```

for the example. First, the theory `movie_rec` containing record type defintions and another helper theory is included (Listing 4.2, line 17). Next, (Listing 4.2, lines 19–21), constants for entity and relationship sets are defined but their value is left unspecified, that is, they remain uninterpreted.

Finally (Listing 4.3), the theory `movie_param_abstract` is instantiated. As a result, the abstract types, projectors, abstract entity sets and constraints between them are all instantiated to use the record types and projectors of of `movie_rec` This completes the definition of the ER model. Note that the only things left unspecified are the uninterpreted attribute types (Listing 4.1) and the uninterpreted entity set constants (Listing 4.2). The correctness of the mapping of the abstract model to the record-based ER model is discussed in Section 6.1.

5. SCHEMA-LEVEL IMPLEMENTATION

The schema level types are flat (non-nested) record types. Sets over schema types are called *tables* in database parlance. All types at this level are concrete primitive types. The choice of what concrete types to use (primitive types such as varchars, integers, etc.) is a design decision that is specific to each schema implementation. In our example, we choose to implement the name and address attribute types as strings. The schema level specification of our example is given in the `movie_schema` theory (Listing 5.1, lines 20–22), which starts by grounding the attribute types. These types define an *interpretation* (lines 24-27) for the unspecified types in `movie_rec`. The choice of primitive types, however, does not affect the specification at this level. Schema types are defined as flat records (Listing 5.2). Note that some of the schema types, like `StudioSchema`, rely on already flat record types. Type refinement across the three levels is summarized in Table 1.

As a design decision, we choose to identify the schema types `UnitOfSchema` and `CrewSchema`. This optimization effectively eliminates the need for a separate `unit_of_table` (Listing 5.3).

Theory	Abstract	ER	Schema
Attributes	-	-	Primitive
Entities	-	Nested	Flat
Relationships	-	Nested	Flat

Table 1: Data type refinement in theories across levels of abstraction. Types are either uninterpreted (denoted by '-'), primitive, flat records, or nested records.

Listing 5.1 (Interpreting Attribute Types).

```
movie_schema: THEORY                                15
  BEGIN                                             16
    IMPORTING props                                 17
    IMPORTING function_results                      18
                                                    19
    NameP:      TYPE = string                       20
    AddressP:   TYPE = string                       21
    NumP:       TYPE = nat                          22
                                                    23
    IMPORTING movie_rec{{                           24
     NameEntity:= NameP,                            25
     AddressEntity:= AddressP,                      26
     NumEntity:= NumP}}                             27
```

Listing 5.2 (Schema Types).

```
StudioSchema: TYPE = StudioEntity                   29
                                                    30
studio_schema_name(                                 31
 s:StudioSchema): NameP = s`name                    32
                                                    33
CrewSchema: TYPE =                                  34
[# num: NumP, studio_name: NameP #]                 35
                                                    36
crew_schema_studio_name_num(                        37
 c: CrewSchema): [NameP,NumP] =                     38
  (c`studio_name, c`num)                            39
                                                    40
UnitOfSchema: TYPE = CrewSchema                     41
```

Listing 5.3 (Table Definitions).

```
studios_table: set[StudioSchema]                    45
crews_table: set[CrewSchema]                        46
unit_of_table: set[UnitOfSchema] = crews_table      47
                                                    48
% Derived Tables                                    49
% --------------                                    50
studio_names_table: set[NameP] =                    51
  image(studio_schema_name,(studios_table))         52
studio_name_crew_nums_table: set[[NameP,NumP]] =    53
```

Listing 5.4 (Key Constraint on *studios_table*).

```
studio_schema_name_injective: AXIOM                62
 injective?[(studios_table), NameP]                63
  (restrict[StudioSchema, (studios_table),         64
    NameP](studio_schema_name))                    65
                                                   66
%studios_entry_for_name: % key                     67
```

Listing 5.5 (Ref. Integrity of *unit_of_table*).

```
unit_of_table_ref_integrity: AXIOM                 87
 FORALL (u: (unit_of_table)):                      88
  member(u`studio_name, studio_names_table)        89
```

Listing 5.6 (Cardinality on *unit_of_table*).

```
studio_for_crew(cr: (crews_table))                 91
 : (studios_table) =                               92
 studios_entry_for_name(cr`studio_name)            93
                                                   94
unit_of: set[[(crews_table),                       95
 (studios_table)]] = graph(studio_for_crew)        96
                                                   97
          function_unit_of:  LEMMA                 98
            function?[(crews_table),               99
             (studios_table)](unit_of)             100
```

Listing 5.7 (Key Constraint on *crews_table*).

```
crew_schema_studio_name_num_injective:             106
 LEMMA                                             107
 injective?[(crews_table), [NameP, NumP]]          108
  (crew_schema_studio_name_num)                    109
                                                   110
% crew_entry_for_studio_num:  % key                111
```

5.1 Constraints

While the constraints on the conceptual ER model are predicates over entity sets, at the schema level, they are encoded as predicates over tables.

The key constraint on `studios_table` (Listing 5.4) axiomatizes the injectivity of `studio_schema_name` projector on `studios_table`. `studios_entry_for_name`, the resulting key function, is obtained like `studio_for_name` is in Listing 3.5.

Referential Integrity of `unit_of_table`: Because `crews_table` and `unit_of_table` are synonymous (Listing 5.3), the referential integrity for `unit_of_table` (Listing 5.5, lines 87–89) needs to specify the constraint only on the studio component of the `unit_of_table`. It is instructive to compare the definition of this constraint at the table level with the constraint `unit_of_ref_integrity` on `unit_of_set` (line 49 of Listing 3.6).

Cardinality constraint on `unit_of_table`: The function `studio_for_crew` (Listing 5.6, lines 91–93) is a composition of the key function `studios_entry_for_name` with the projector derived from the `studio_name` field. The cardinality constraint of `unit_of_table` is thus automatically satisfied (Listing 5.6, lines 98–100).

Key Constraints of `crews_table`: the projection function `crew_schema_studio_name_num` (Listing 5.2) is injective on `crew_table` because `CrewSchema` is defined in terms of a studio name and a number attribute.

Constraint	PVS Specification
1	`studio_name_injective_on_studios_set`: AXIOM (Listing 3.5) `studio_schema_name_injective`: AXIOM (Listing 5.4)
2	`crew_studio_num_injective_on_crews_set`: AXIOM (Listing 3.8) `crew_schema_studio_name_num_injective`: LEMMA (Listing 5.7)
3	`function_unit_of`: AXIOM (Listing 3.7) `function_unit_of`: LEMMA (Listing 5.5)
4	`unit_of_ref_integrity`: AXIOM (Listing 3.6) `unit_of_table_ref_integrity`: AXIOM (Listing 5.5)

Table 2: Specification of constraints across `movie` theories. For each row, the entry in the left cell refers to the constraint in English in Section 2.1. For the right cell, the upper entry is the constraint in the abstract model (Section 3) and the lower entry is the constraint in the schema model (Section 5).

.

Table 2 summarizes the different constraints of the mini-example. The constraints are specified at three levels: natural language (Section 2), and PVS specification in the abstract model and the schema-level model. Because of representation decisions made at the schema level (namely, identifying the implementation of `unit_of` table with that of the `crews_table`, some constraints expressed as axioms at the abstract level are lemmas at the schema level. In addition, the axiom `unit_of_table_ref_integrity`, combined with the equivalence of representation between `unit_of_table` and `crews_table` is strong enough to implement the axiom `unit_of_ref_integrity`, the integrity constraint for the `unit_of` abstract entity set.

6. THEORY INTERPRETATIONS AND THE CORRECTNESS OF MAPPING

We have seen how to specify a data model at three levels of abstraction. How are these models related, and in what sense is a data model valid with respect to another? We rely on PVS's notion of implementation between theories [31]. A data model A is *valid* with respect to a model B if the theory specifying model B provides an *implementation* of the theory specifying A. When A is valid with respect to B, we say there is a valid mapping from A to B. PVS has two separate, but related notions of implementation: instantiation and interpretation. Both of these are specified using the IMPORT keyword and used in the example specification.

6.1 Theory instantiation and abstract to ER

In PVS, a parametric theory A may be *instantiated* by a theory B using an 'IMPORT A' statement in B. This supplies actual arguments to the parametric types and constants of A. All of A's parameterized definitions and theo-

rems are available as instances in B, with the actual arguments to the parameters supplied in IMPORT statement in B. For B to correctly implement A, however, all the type correctness conditions, if any, generated by the IMPORT must be proved.

When `movie_param_abstract` is instantiated in the theory `movie_er` (Listing 4.3), no TCCs are generated. This is not entirely unexpected, since the record types and entity sets at the ER model level are obtained by a direct instantiation of the corresponding parameters at the abstract level. This establishes the correctness of the mapping from `movie_param_abstract` to `movie_er`.

6.2 Theory interpretation and ER to schema

In PVS, a theory A containing types, constant definitions, axioms and theorems may be *interpreted* by theory B if B provides an interpretation for the uninterpreted types and constants of A in such a way that the axioms of A may be interpreted as theorems in B. B thus becomes an "implementation" of A, demonstrating A's consistency with respect to B, provided the TCCs generated by the IMPORT in B of theory A are all proved.

To show that the schema model correctly interprets the ER model, we need to construct an interpretation for the uninterpreted types, constants (entity sets) and also prove as theorems the axioms in the ER model. This requires some effort since the schema model and the ER model operate at different but non-abstract type levels: ER models with nested records, and schema models with flat records.

Finally, the only uninterpreted objects in `movie_er` are the attribute types by virtue of importing `movie_rec`, and the entity sets (Listing 4.1). The schema model provides an interpretation for the attribute types (IMPORT statement in Listing 5.1). The parameter list to the import is a mapping *uninterpreted-constant* := *interpreted-value*. Next, we see how the interpretation of entity sets is constructed.

6.3 Entity construction

To build an interpretation for entity sets, we start by constructing an interpretation of the *entity elements* of the ER model using *entries*, which are elements of the tables in the schema model. Entity construction is done by defining a set of functions that construct an entity from a table entry (Listing 6.1). These functions are then used to interpret the entity sets (Listing 6.2) of the ER model. Recall that these entity sets were defined as uninterpreted constants in the ER model. Finally, the `IMPORT` statement of PVS is used to create an interpretation of the ER model's entity sets in terms of the tables in the schema model (Listing 6.3). Figure 2 illustrates the different notions of implementation (importing) used amongst the PVS theories in our example models.

6.4 Verifying type correctness conditions

The typechecking of the specifications and import statements in PVS automatically generates type correctness conditions. The theories `movie_param_abstract` and `movie_schema` generate one and five TCCs respectively. The theories `movie_rec` and `movie_er` generate no TCCs. The library theories (not shown) together generate three TCCs. None of these proofs are difficult to do. The completion of the proofs of the TCCs implies that the mapping between the ER model and the schema level is sound. Proof statistics for the PVS specifi-

Listing 6.1 (Entity Construction).

```
studio_instance_for_entry
 (s:(studios_table)):  StudioEntity = s

crew_instance_for_entry
 (c: (crews_table)): CrewEntity =
  LET n = c`num, sn = c`studio_name IN
   LET  se =
     studios_entry_for_name(sn)
    IN LET st =
     studio_instance_for_entry(se)
       IN (# num:= n, studio:= st #)

unit_of_instance_for_entry
 (u: (unit_of_table)): UnitOfEntity =
  LET cr = crew_instance_for_entry(u)
    IN LET st = crew_entity_studio(cr)
      IN (# crew:= cr, studio:= st #)
```

Listing 6.2 (Entity Sets from Tables).

```
studio_instances_set: set[StudioEntity] =
   studios_table

crew_instances_set: set[CrewEntity] =
 image(crew_instance_for_entry,
       crews_table)

unit_of_instances_set: set[UnitOfEntity] =
 image(unit_of_instance_for_entry,
       unit_of_table)
```

Listing 6.3 (Interpreting *movie_er*).

```
IMPORTING movie_er{{
 studios_entity_set := studio_instances_set,
 crews_entity_set   := crew_instances_set,
 unit_of_entity_set :=
  unit_of_instances_set}}
END movie_schema
```

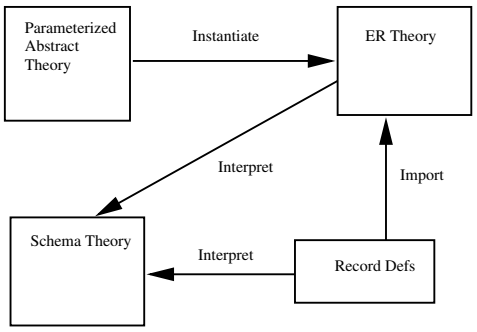


Figure 2: A high level view of the import relationships between different theories implementing the movie example. (Not all theories used are shown.)

Theory	Lines	TCCs	Lemmas
props	7	0	1
function_results	21	0	4
key	18	3	1
movie_rec	45	0	0
movie_er	32	0	0
movie_param_abstract	133	4	3
movie_schema	308	15	2
Total	**564**	**22**	**11**

Table 3: TCCs and user formulas in the different theories used to implement the complete movie example [3].

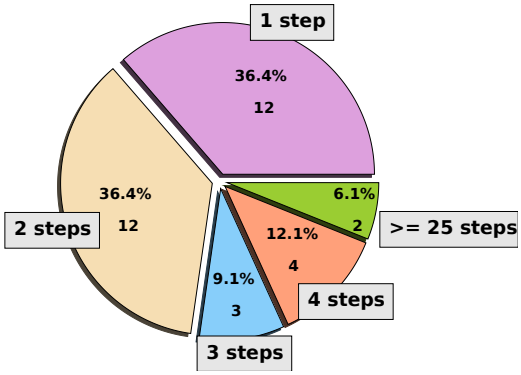


Figure 3: Distribution of the 33 proofs for the implementation of the full movie data model of Section 2 according to size (in number of user proof steps). All but two of the proofs are four steps or less.

cation of the complete Ullman and Widom example [3, 12] are shown in Section 7.

7. RESULTS

The number of lines of code, the number of TCCs generated, and the number of user formulas in each of the seven theories constituting the specification of the complete movie example [3, 12] are shown in Table 3. The abstract and schema specifications make up the bulk of the source code (441 lines out a total of 564). A total of 22 TCCs are generated. These are divided amongst the abstract and schema specifications, and the key library theory. The rest of the theories do not generate any TCCs, including `movie_er`. There is, on an average, about one tcc generated for every 30 lines of code. The specification also consists of 11 user-defined lemmas. Together with the TCCs, the total number formulas that need to be proved is 33. Not surprisingly, the bulk of the TCCs generated are for the theory `movie_schema` (15 of 22).

The distribution of the sizes of proofs of these 33 formulas is shown in Figure 3. All but two of them are of four or less steps in length and almost three-fourths are of length two or less. Fortunately, the two lemmas with much longer proofs (25 and 47 steps) are independent of the example model; they belong to library theories.

The results of Figure 3 encourage us to speculate that even as the number of model-specific constraints increase, the number of TCCs will increase, but not the sizes of their proofs. We expect that the number of generated TCCs to be proportional to the number of constraints in the model. We assume that the arity of relationships and the number of attributes on an entity is bounded. This implies that number of constraints varies linearly as the size of the ER diagram. This leads us to conjecture that the number of TCCs generated is at most linear in the size of the ER diagram of the model.

The proofs in our implementation all use only elementary proof steps and PVS's built-in strategies like GRIND. User-defined PVS proof strategies have not been used. Their use could further reduce the size of some of the longer proofs.

8. RELATED RESEARCH

Formalizing conceptual models for database applications was the original motivation for Codd's relational model and the conceptual ER model of Chen, which are both based on the theory of sets and relations [11, 13]. The relatively more recent object-oriented models [9] and object-relational models [32] also employ formal notations for their presentation.

Languages like Datalog are popular with the logic programming and deductive database community [10, 19]. Neumann and others use Datalog for building a framework for reasoning with data models [23, 30]. This approach relies on encoding instances, models and metamodels as Datalog programs. Integrity constraints are encoded as predicates and verification is done by querying these predicates for violations. However, Datalog is a highly restricted variant of Prolog and as such is only slightly more powerful than relational algebra and relational calculus, which form the core of the dominant databse query language, SQL. This restriction is because databases are often so large that even quadratic evaluation times are unreasonable. Datalog evaluations are thus explicitly decidable while PVS type-checking is not.

There are many papers recognizing the need for a formal approach to data modeling [6, 8, 33, 34]. This is also the case with modeling in related areas like object oriented software engineering and UML[7, 38]. There has also been work on the importance of conceptual models in the context of development [25], in business processes and business

intelligence [29], and in decision support systems [24].

Extensions to the ER model have been proposed with reasoning, semantics and constraint specification features [15]. Constraint specification has also been considered in the context of object-oriented databases and UML [21]. A generic specification process of diagram languages such as the ER model has also been reserached [28]. Specification languages are common in knowledge-based systems [17] and semantic databases [5]. Conceptual model-based verification and validation have also been researched in the context of specific applications such as diagnosis [36]. UML and OCL have been modeled formally using PVS [26]. In the ontology space, PVS has been used to formalize OWL and ORL specifications [14]. More recently, Mandelbaum et al. have proposed a data domain language PADS/ML for building a uniform automatic compilation framework of data models for data in ad hoc formats [27]. Their approach is promising, but is focused towards data analysis, and not modeling per se.

9. FUTURE WORK

This work is an initial step in the building of (semi) automated frameworks based on formal specification of data models. There are several directions for future work:

Automation: It should be relatively straightforward to automatically generate the PVS specification from an ER diagram. The second aspect of the automation involves generating automatic proofs of type correctness conditions and the correctness lemmas. Since most of the proofs involved a few steps, we expect that it should be possible to automate most, if not all of the proofs. This is a positive indication for building future tools based on this methodology.

Scaling: We have explored the approach with a small, text book example with about 12 constraints. Industry scale data modeling includes hundreds of constraints between dozens of entities and relationship types. We plan to use data models from industry case studies to investigate how our approach scales. The success of this scaling will be heavily dependent on the level of automation that can be achieved in generating the proofs of correctness and TCCs.

Trigger generation: Triggers are the practical implication of constraints. It should be possible to automatically translate constraints into triggers, which are tests that ensure the invariants are maintained at the end of every update to the database. However, while constraints are typically stated in terms of global properties, an efficient trigger should involve computation proportional to the size of the update to the database, not the size of the database itself.

Impact on design exploration: Model verification has an important role in allowing the designer to explore various design options during the modeling phase. in each case, the verification framework ensures that the design is explored within the boundaries of correctness. We plan to investigate how our framework supports such a correct-by-construction design methodology.

Working with other models: We have applied the specification language approach to traditional entity-relationship models of data. It should be interesting to consider formal specification of data models, like object models and their mapping to relations.

10. CONCLUSIONS

We have shown how data models may be specified and reasoned within PVS at different levels of abstraction. In particular, we have demonstrated how the support for higher-order functions, type checking, and interactive theorem proving in PVS allows the data modeler to reason about the interactions between the various data constraints. These are usually harder to do when using ER diagrams alone.

While design verification plays an important role in other disciplines (hardware and program verification), it has generally received less attention in ER data modeling. We believe this is due to the limited use of standard, formal notations and languages with verification support to express reasoning about data models. The work presented in this paper is a demonstration that general purpose specification languages like PVS with their powerful typechecking support can fill this gap.

Acknowledgements: We thank Sam Owre of SRI for patiently answering many of our queries on PVS.

11. REFERENCES

[1] The B-method. http://vl.fmnet.info/b/. Visited October 2007.

[2] PVS: Prototype Verification System. http://www.csl.sri.com/pvs. Visited October 2007.

[3] PVS source code accompanying [12] and this paper. http://www.iiitmk.ac.in/~choppell/research/code/movie-data-model/index.html Visited October 2007.

[4] The Z notation. http://vl.zuser.org/. Visited October 2007.

[5] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12:525–565, 1987.

[6] G. D. Battista and M. Lenzerini. Deductive entity-relationship modeling. *IEEE Trans. Knowl. Data Eng.*, 5(3):439–450, 1993.

[7] R. Breu, U. Hinkel, C. Hofmann, C. K. B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In M. Aksit and S. Matsuoka, editors, *Proceedings of ECOOP'97 – Object Oriented Programming. 11th European Conference*, volume 1241 of *LNCS*, pages 344–366. Springer, 1997.

[8] D. Calvanese, M. Lenzirini, and D. Nardi. *Logics for Databases and Information Systems*, chapter Description Logics for Conceptual Data Modeling. Kluwer Academic, 1998.

[9] R. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

[10] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases (Surveys in Computer Science)*. Springer, 1990.

[11] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–37, March 1976.

[12] V. Choppella, A. Sengupta, E. Robertson, and S. D. Johnson. Constructing and Validating Entity-Relationship models in the PVS Specification Language: A case study using a text-book example.

Technical Report 632, Indiana University Computer Science, April 2006.

[13] E. Codd. A relational model for large shared data banks. *Communications of the ACM*, 6(13):377–387, June 1970.

[14] J. S. Dong, Y. Feng, and Y. F. Li. Verifying OWL and ORL ontologies in PVS. In Z. Liu and K. Araki, editors, *1st International Colloquium on Theoretical Aspects of Computing (ICTAC) 2004*, volume 3407 of *LNCS*, pages 265–279. Springer, 2005.

[15] G. Engels, M. Gogolla, U. Hohenstein, K. Hulsmann, P. Lohr-Richter, G. Saake, and H.-D. Ehrich. Conceptual modeling of database applications using extended ER model. *Data Knowledge Engineering*, 9:157–204, 1992.

[16] L. M. G. Feijs. Norman's database modularized in COLD-K. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II - theory, tools and applications*, volume 490 of *LNCS*, pages 205–229. Springer, 1991.

[17] D. Fensel. Formal specification languages in knowledge and software engineering. *The Knowledge Engineering Review*, 10(4), 1995.

[18] J. Fitzgerald and C. Jones. Modularizing the formal description of a database system. In C. H. D. Bjorner and H. Langmaack, editors, *VDM '90: VDM and Z - Formal Methods in Software Development*, volume 428 of *LNCS*, pages 189–210, 1990.

[19] H. Gallaire, J. Minker, and J.-M. Nicolas. Logic and databases: A deductive approach. *ACM Comput. Surv.*, 16(2):153–185, 1984.

[20] C. George. The NDB database specified in the RAISE specification language. *Formal Aspects of Computing*, 4(1):48–75, 1992.

[21] M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.

[22] D. Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[23] N. Kehrer and G. Neumann. An EER prototyping environment and its implemetation in a datalog language. In G. Pernul and A. M. Tjoa, editors, *Entity-Relationship Approach - ER'92, 11th International Conference on the Entity-Relationship Approach, Karlsruhe, Germany, October 7-9, 1992, Proceedings*, volume 645 of *Lecture Notes in Computer Science*, pages 243–261. Springer, 1992.

[24] R. Kimball. Is ER modeling hazardous to DSS? *DBMS Magazine*, October 1995.

[25] C. H. Kung. Conceptual modeling in the context of development. *IEEE Transactions on Software Engineering*, 15(10):1176–1187, 1989.

[26] M. Kyas, H. Fecher, F. S. de Boer, J. Jacob, J. Hooman, M. van der Zwaag, T. Arons, and H. Kugler. Formalizing UML models and OCL constraints in PVS. In *Proceedings of the Semantic Foundations of Engineering Design Languages (SFEDL '04)*, pages 39–47, 2005.

[27] Y. Mandelbaum, K. Fisher, D. Walker, M. Fernandez, and A. Gleyzer. PADS/ML: A Functional Data Description Language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 77–83. ACM Press, January 2007.

[28] M. Minas. Specifying diagram languages by means of hypergraph grammars. In *Proc. Thinking with Diagrams (TwD'98)*, pages 151–157, Aberystwyth, UK, 1998.

[29] L. Moss and S. Hoberman. The importance of data modeling as a foundation for business insight. Technical Report EB4331, NCR, November 2004.

[30] G. Neumann. Reasoning about ER models in a deductive environment. *Data and Knowledge Engineering*, 19:241–266, June 1996.

[31] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI International, April 2001.

[32] M. Stonebraker and D. Moore. *Object Relational DBMSs: The Next Wave*. Morgan Kaufmann, 1995.

[33] A. ter Hofstede and H. Proper. How to formalize it? formalization principles for information systems development methods. *Information and Software Technology*, 40(10):519–540, 1998.

[34] B. Thalheim. *Entity-Relationship Modeling: Foundations of Database Technology*. Springer-Verlag, 2000.

[35] J. D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 2 edition, 2002.

[36] F. van Harmelen and A. ten Teije. Validation and verification of conceptual models of diagnosis. In *Proceedings of the Fourth European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV97)*, pages 117–128, Leuven, Belgium, 1997.

[37] A. Walshe. *Case Studies in Systematic Software Development*, chapter NDB: The Formal Specification and Rigorous Design of a Single-User Database System. Prentice Hall, 1990.

[38] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise modeling with UML*. Object Technology Series. Addison Wesley, 1998.

[39] N. Winterbottom and G. Sharman. NDB: A non-programmer database facility. Technical Report TR.12.179, IBM Hursley Laboratory, England, September 1979.

Topology in PVS

Continuous Mathematics with Applications

David R Lester
School of Computer Science, Manchester University
Manchester M13 9PL, United Kingdom
dlester@cs.man.ac.uk

ABSTRACT

Topology can seem too abstract to be useful when first encountered. My aim in this paper is to show that – on the contrary – it is the vital building block for continuous mathematics. In particular, if a proof can be undertaken at the level of topology, it is almost always simpler there than when undertaken within the context of specific classes of topology such as those of metric spaces or Domain Theory.

Categories and Subject Descriptors

MSC 54 [**General Topology**]: Topological Spaces; MSC 03B35 [**General Logic**]: Mechanization of Proofs and Logical Operations

Keywords

Continuous Mathematics, Probability, Programming Language Semantics, Proof Assistants, PVS, Theorem Provers, Topology

1. INTRODUCTION

Topology can seem too abstract to be useful when first encountered. My aim in this paper is to show that – on the contrary – it is the vital building block for continuous mathematics. In particular, if a proof can be undertaken at the level of topology, it is almost always simpler there than when undertaken within the context of specific classes of topology such as those of metric spaces or Domain Theory.

And why might continuous mathematics be of interest to proponents of formal methods? This is a hard question for me as a relative outsider to the field to answer. However, I would suggest that although proofs about the internal workings of programs can – and should – be framed in terms of discrete mathematics, when it comes to formal requirements capture, having some credible model of the outside world with which the computer system is supposed to interact is vital. If that outside world is continuous – as it inevitably is for control systems – how else do we show that the program works properly, other than by showing that it interacts with the relevant differential equations correctly? Other than the use of proof assistants, there is nothing new in this approach: this is the same technique we used at Hawker-Siddeley validating 8080-based control systems in the 1970s. The completed system was connected directly to an analog computer simulating the operating environment, many properties of the assembly code having already been proven correct using pencil and paper.

I will discuss two particular applications that I've implemented on top of the topology library: probability theory and compiler correctness. Only the second of these areas has been properly developed, but enough progress has been made to be confident of the outcome in the first case. An abortive attempt to deal with differentiation in an arbitrary Hilbert Space will be obliquely alluded to.

So, perhaps the most useful way to read this paper, is as a *précise* of the "Pitfalls I've fallen into, and how to avoid them" kind. They'll be plenty of them, I'm afraid.

2. WHAT IS A TOPOLOGY?

A topology is a set of *open sets*, obeying certain conditions on the open sets. Before listing these conditions, it's probably best to give the motivating definition for topology: that of continuity.

Definition 1. A function $f : T_1 \to T_2$ is *continuous* if, and only if, the inverse image of every open set in T_2 is an open set in T_1.

It is now immediate that the identity function is continuous, as is the composition of two continuous functions. Compare the composition proof to the equivalent metric space proof given in Appendices A and B. Notice that the instantiations in the topological proof could be easily mechanized, whilst the skill and judgement required to correctly instantiate in the metricized proof – not to mention the precise ordering in which we interleave instantiation and witnessing – make for a rather hard mechanization process. And then, when we consider continuous functions between Scott Topologies (which are not metricizable), we have to perform yet another proof that the composition of two (Scott-)continuous functions will be (Scott-)continuous.

What constraints are there in picking the open sets? Firstly, we insist that both the empty and full sets are open. Next,

AFM'07, November 6, Atlanta, GA, USA.

that the union of any collection of open sets is open and finally that the intersection of any two open sets is open. And that's it! So, why am I insisting that this is such a big deal? Quite simply, if you are dealing with continuous mathematics, then this is the fundamental definition that you are using, even if it is often well hidden.

Instead of attempting to convince you of the beauty of the theory, I'll instead provide a few examples of practical applications: in particular general probability theory, and denotational semantics. Before we do this, it will be as well to look at more restrictive topologies.

Definition 2. A topology is *Hausdorff* if given distinct x, y, there exist disjoint open sets U and V containing x and y respectively.

We note that every metric space (of which more later) is Hausdorff, and that in a hausdorff space every convergent sequence of points has a unique limit.

Another useful concept is that of compactness. With this property we make induction available as a proof technique for any compact set.

Definition 3. A set A is *compact* if for every open cover U there is a finite $V \subseteq U$ and V is also an open cover. U is an open cover of A if each element of U is an open set and $A \subseteq \bigcup U$.

The *Bourbaki-istes* ([6]) amongst you may wish to argue that the above definition is that of quasi-compactness, reserving the term "compact" for a quasi-compact set in a hausdorff space. Gratifyingly, such petty distinctions cause much more trouble than they are worth in PVS.

We will also be interested in methods to generate a topology from a *base* (or *basis*) set. As an example, consider how we can use the open intervals $(a,\ b)$ to generate any open set in the usual topology for $\mathbb{R}$. We will be particularly concerned with those topologies that have a countable base: such topologies are called *second countable*. The usual real topology is countably-based as the rational open intervals constitute such a base.

As part of the proof of the generalized Heine-Borel Theorem (see later), we take the opportunity to show that in a second countable topological space we can reduce the covering problem to a merely countable one (the proof consists of 648 lines in lindelof.prf).

Theorem 1 (Lindelöf's Covering Theorem). *In a second countable topological space every open cover of the space has a countable subcover.*

I'm not entirely convinced that the following should live within the topology library, but two other topics are needed to define measure theory: σ-algebras and Borel sets and functions. A σ-algebra contains the empty set and is closed under the operations of taking complements and countable unions. The smallest σ-algebra over T therefore consists of just the empty set and the fullset T; the largest is the power set $\mathcal{P}(T)$. Note carefully, the power set is "too big" to be used when the cardinality of T is uncountable.

The Borel sets of a topological space are then the members of the σ-algebra generated by the open sets of the topology. As an example, consider the Borel sets of $\mathbb{R}$. Every open, closed, semi-open and semi-closed interval in $\mathbb{R}$ is a Borel set, along with countable unions of such intervals.

Finally, we define *Borel functions*.

Definition 4. A function $f : T_1 \to T_2$ is *Borel* if, and only if, the inverse image of every Borel set in T_2 is a Borel set in T_1.

It is a matter of moments (20 lines of .prf) to prove that every continuous function is also a Borel function.

2.1 Summary

The library of results for general topology has 202 theorems, the proof of which requires 15,654 lines in the .prf files. This library has been released. The textbook used is largely [22] supplemented by material in [2, 3].

3. PROBABILITY

One possible approach to probability is to distinguish continuous and discrete random variables. Initially, this appears to work. However, as the theorems become deeper (and more general) the distinction becomes ever more irritating. It is also worth bearing in mind that there are random variables that are neither continuous nor discrete; whether they have a practical use other than as counter-examples is a moot point.

The target I have set myself is to prove the Central Limit Theorem.

Theorem 2 (Central Limit Theorem). *Let X_1, X_2, ... X_n be a sequence of independent identically-distributed random variables with finite means μ and finite non-zero variances σ^2 and let $S_n = X_1 + X_2 + \cdots + X_n$. Then*

$$\frac{S_n - n\mu}{\sqrt{n\sigma^2}}$$

converges in distribution to a random variable that is distributed $N(0,1)$ as $n \longrightarrow \infty$.

I intend to prove this by manipulating the characteristic functions of the X_n; with enough machinery in place, this appears to be relatively straightforward. However, as we shall see, providing sufficient machinary necessitates a considerable amount of preliminary work.

First, a definition. The 3-tuple $(\Omega,\ \mathcal{F},\ \mathbb{P})$ is a *Probability Space* under the following conditions.

Definition 5. A *Probability Space* has three components: a *Sample Space* (Ω); a σ-algebra of permitted events ($\mathcal{F}$);

and a probability measure $\mathbb{P} : \mathcal{F} \to [0,1]$. There are obviously constraints on the probability measure: we require $\mathbb{P}(\{\}) = 0$, $\mathbb{P}(X^c) = 1 - \mathbb{P}(X)$, and that if the X_n are pairwise-disjoint, then $\mathbb{P}(\bigcup_{n=0}^{\infty} X_n) = \sum_{n=0}^{\infty} \mathbb{P}(X_n)$.

Definition 6. The characteristic function of a random variable X is the function $\phi : \mathbb{R} \to \mathbb{C}$ given by:

$$\phi(t) = \mathbb{E}(e^{itX})$$

At once our problems begin: should we use the complex numbers provided in the NASA libraries? (No: using this library causes the PVS automatic strategies involving commutativity and associativity to be switched-off — even for real-valued expressions.) We note that unless we wish to prove this result twice (once for continuous random variables and again for discrete ones), we will need to define the expectation operator $\mathbb{E}$ in terms of the underlying probability measure $\mathbb{P}$ associated with the random variable X:

$$\mathbb{E}(X) = \int_{\omega \in \Omega} X(\omega) d\mathbb{P}.$$

This means that the characteristic function is defined as the *Lebesgue-Stieltjes Integral* of $e^{it\omega}$ with respect to the probability measure $\mathbb{P}$:

$$\phi(t) = \int_{\omega \in \Omega} e^{it\omega} d\mathbb{P}.$$

The link to Fourier Convolutions is now clear. At this point work on the probability library ceased, until such time as the Fourier Transform material became available.

As Reimann integration is available to us in the NASA analysis library, why don't we use that instead? The first problem is that we would need to extend the material to deal with indefinite integrals (look carefully at the definition of characteristic functions above to see why). Consider the attempted "definition" of indefinite integral as the limit of the following sequence of definite integrals:

$$\int_{-n}^{n} f dx \longrightarrow \int_{-\infty}^{\infty} f dx,$$

as $n \longrightarrow \infty$. If f is the identity function then each integral on the left is 0, but the right-hand side is undefined. Care is clearly required.

The second clue that we will need to adopt a more sophisticated approach to integration is the presence of a σ-algebra in the definition of a probability space. One might also consider the name of the third component as a "probability measure" a big hint. Although we could persist with Riemann integration, many theorems would be less general than they might be and theorems characterizing the density functions become impossible to prove.

The high point of the material already undertaken is probably Bayes' Theorem on conditional probability.

THEOREM 3 (BAYES' THEOREM). *Provided B is non-null, and $\bigcup_{i=0}^{n} A_i = \Omega$, then for $0 \leq j \leq n$, the conditional probability of A_j given B is:*

$$\mathbb{P}(A_j \mid B) = \frac{\mathbb{P}(B \mid A_j)\mathbb{P}(A_j)}{\sum_{i=0}^{n} \mathbb{P}(B \mid A_i)\mathbb{P}(A_i)}$$

This constitutes a mere 100 lines of proof in conditional.prf.

3.1 Summary

A fair amount of preliminary work has already been undertaken (Bayes Theorem for example). The library for probability currently has 62 theorems, the proof of which requires 10,985 lines in the .prf files. This library is not yet linked with the measure theory and is consequently not ready for release. Indeed, with PVS 4.0 there are problems saving the pvscontext in the measure theory library that are a cause for concern. Most of the material comes from [12], though some of the more general material on products of probability spaces comes from [13].

There is currently nothing in this library that is not already in Hurd and Nedzusiaks' work [14, 16, 17].

4. MEASURE THEORY

Having considered and disgarded Riemann Integration to describe probabilities, we are forced to consider doing a "proper job". How bad can it be?

Firstly, we note that we will almost certainly wish to deal with sets of measure ∞, so our measure functions $\mu : \mathcal{F} \to \overline{\mathbb{R}}_{>0}$. Having learnt the lesson of a direct implementation of the complex numbers, this time we are explicit in the use of extended non-negative reals, and do not attempt to embed them within the PVS number_field library. The somewhat grubby use of `x_add` for the addition operator on the extended non-negative reals ($\overline{\mathbb{R}}_{>0}$) works perfectly, in the sense that it does not cause the switch-off of the default strategies for the reals that we will subsequently require.

The other general library concerns properties of summation over sets. Anticipating the need to use this function for discrete probability, I made it capable of handling summation over countable sets (provided that the image under the function was absolutely convergent). What I forgot (*pace* Hilbert Space) is that provided only countably many elements of the set are non-zero then once more we merely require absolute convergence of the non-zero elements of the image. I propose that we modify my sigma_set.pvs so that a set is "permissible" precisely when it is: finite, or when the non-zero elements of its image are absolutely convergent (we require absolute convergence so that the summation may be attempted in any order). This generalization could replace the finite version currently available in the NASA library.

We now define the *Measure Spaces* within which integration can take place.

Definition 7. A measure space is a 3-tuple (T, S, μ), where S is a σ-algebra over T, and $\mu : S \to \overline{\mathbb{R}}_{>0}$ satisfies:

- $\mu(\{\}) = 0$; and

- $\sum_{n=0}^{\infty} \mu(X_n) = \mu(\bigcup_{n=0}^{\infty} X_n)$ for all collections X which are pairwise disjoint.

For the *cogniscenti*, we now present two theorems that require a great deal of effort, and yet are treated very lightly in the text books. We begin with Berberian Theorem 4.1.20.

THEOREM 4. *If u is a sequence of measurable functions that converges pointwise to $f : T \to \mathbb{R}$, then f is measurable.*

This took a mere 537 lines to prove in measure_space.prf. But it remains conceptually one of the trickier proofs to perform.

The next result is "trivial" and "obvious", which I'm rapidly learning to treat as a synonym for "boring", "long-winded" and generally "not worth the effort". (This is Berberian Theorem 4.3.7).

THEOREM 5. *If P is a finite partition of the integrable step function i, then $\int id\mu = \sum_{p \in P} i(x \in p).\mu(p)$*

The import of this theorem is that it does not matter how we divide up an integrable step function into a sum of characteristic functions, the integral remains the same. Keen students of mathematics may wish to attempt a shorter proof of this result. At the present time I wish to submit this proof as a candidate for the longest single proof attempted in PVS. 9606 lines of isf.prf are taken up by the proof. Other than that, the theorem is completely ignorable: it is merely an induction argument over the cardinality of the finite set P.

I regard the following theorems as the highlights of the current development.

THEOREM 6. *If f is integrable, and h is a measurable function with $|h| \leq M$ almost everywhere, then fh is integrable and*

$$\int |fh| d\mu \leq M \int |f| d\mu.$$

THEOREM 7. *For two integrable functions f and g, and E measurable:*

$$f = g \text{ a.e. if, and only if } \forall E.\ \int_E f d\mu = \int_E g d\mu$$

Most of the effort in proving these theorems lies in setting up the framework, rather than anything intrinsically difficult in the theorems themselves.

The next few targets in this library are obvious: firstly, prove Beppo-Levi's Monotone Convergence Theorem, Lebesgue's Dominated Convergence Theorem, and Fatou's Lemma. This should be a day's work. Next, since product measures are by and large completed, work on the Hahn-Kolmogorov Extension Theorem, before turning attention to the Tonelli-Fubini Theorems about the manipulation of iterated integrals.

The next big piece of work is to prove Lebesgue's Criterion for Riemann Integrability. Without this result (that, for "sensible" functions, the two integration methods agree), we are unable to perform much practical integration. Recall that the easy way to compute an integral is to differentiate and then use the "Fundamental Theorem of Calculus". The Lebesgue equivalent is much trickier to use and fiendish to define and prove (it uses Dini-derivatives to handle the derivative at points of discontinuity).

Nevertheless, Lebesgue's Criterion for Riemann Integrability is a vital component in the next target: proving that Fourier Transforms exists and have the properties we require. This will be grubby and time-consuming, but forms a bridge to the definition and calculation of a characteristic function for a Random Variables.

Apart from a few basic properties of sequences and series of real numbers, we appear to be largely divorced from the fundamentals of topology.

Until – that is – we attempt to prove the following fundamental property of the Lebesgue Outer Measure λ^*.

Definition 8. If we represent the *length* of an interval I as $\lambda(I)$, then the *Lebesgue Outer Measure* of a set A, denoted $\lambda^*(A)$, is defined as:

$$\lambda^*(A) = inf \left\{ \sum_{n=0}^{\infty} \lambda(I_n) \mid A \subseteq \bigcup I_n \right\}$$

where I_n ranges over all possible sequences of open intervals.

LEMMA 8. *For any bounded interval I:*

$$\lambda^*(I) = \lambda(I)$$

For this, we need Heine-Borel (either over the usual topology for $\mathbb{R}$, or preferably for $\mathbb{R}^n$).

4.1 Summary

The library for measure theory currently has 590 theorems, the proof of which requires 121,916 lines in the .prf files. This library is releasable, but would be much improved with the convegence theorems, Riemann-Lebesgue Criterion, Tonelli-Fubini and Convolution Theorems incorporated. Some of the easier material derives from [23], but the majority comes from [3]. Product measure spaces are dealt with by Halmos in [13].

Whilst the material defining Measures and Measure Spaces contains little that is not already present in Hurd and Bialas' work [14, 4, 5], the material on measure theoretic integration is potentially something new in the realm of PVS theorem proving.

5. METRIC SPACES

Initially, I thought that relatively little would be required in this library; much of the action would take place either above (in topology) or below (in normed space). It turns out I was wrong. We need to show two key results: that the reals are complete, and the Heine-Borel Theorem.

Definition 9. The function $d : T, T \to \mathbb{R}_{>0}$ is a *metric*, if

$$\begin{array}{lcl} d(x,y) & = & 0 \text{ if, and only if, x=y;} \\ d(x,y) & = & d(y,x); \\ d(x,z) & \leq & d(x,y) + d(y,z). \end{array}$$

If in addition every cauchy sequence is convergent, then the *metric space* (T, d) is said to be *complete*.

Definition 10. A set X is *totally bounded* if, for every $r > 0$, there is a finite set of open balls of radius r covering the set X.

THEOREM 9 (HEINE-BOREL (GENERALIZED)). *In a complete metric space, a set is compact if, and only it it is totally bounded.*

Much of this has been proved, but the proof is incomplete. The following special case *has* been proved.

THEOREM 10 (HEINE-BOREL (FOR $\mathbb{R}$)). *The closed interval $[a, b] \in \mathbb{R}$ is compact.*

The key preliminary result (that the real topology is complete) had already been proved in the analysis library. One other pretty result is the equivalence of continuity and uniform continuity in a compact space.

THEOREM 11. *In a compact space, a function is continuous if, and only if, it is uniform continuous.*

The proof of ($\Rightarrow$) exploits the compactness to reduce the problem to a finite case, and then uses induction to complete the proof. The proof of ($\Leftarrow$) is trivial.

5.1 Summary

The library for metric spaces currently has 86 theorems, the proof of which requires 15,730 lines in the .prf files. I am in the middle of adjusting the interface to this library; it is therefore not ready for release. Material in this library comes primarily from [22], augmented by results in [2, 3].

Much of the material in this library is covered in the pre-existing NASA analysis library. However, the proposed new library generalizes the current version extensively. For example, with the current NASA/PVS library, it is not possible to state that the $\tan(x)$ function is continuous on the open set $X = \{x \mid x \neq \pi(k + \frac{1}{2})\}$; with the new library such statements are perfectly acceptable. This would ease the use of the properties of trigonometric functions in the NASA/PVS trig library considerably. Obviously, defining the metric over an arbitrary type rather than a connected subset of the reals also has the potential to extend the applicability of the library.

6. PROGRAMMING LANGUAGE SEMANTICS

At first it might seem strange that we need continuous mathematics to describe the behaviour of a computer program: after all, aren't computer programs notorious for their sensitivity to even the smallest errors? The insight is that a looping computer program is (hopefully) building a better and better approximation to the desired answer.

The denotational semantics of a simple `While` language is defined.

$$\begin{array}{lcl} \mathcal{S}[\![x := a]\!]\ s & = & s[x \mapsto \mathcal{A}[\![a]\!]\ s] \\ \mathcal{S}[\![\texttt{skip}]\!]\ s & = & s \\ \mathcal{S}[\![S_1;\ S_2]\!] & = & \mathcal{S}[\![S_2]\!] \circ \mathcal{S}[\![S_1]\!] \\ \mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!] & = & \text{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}[\![S_1]\!],\ \mathcal{S}[\![S_2]\!]) \\ \mathcal{S}[\![\texttt{while } b \texttt{ do } S]\!] & = & \text{fix}(F) \\ \multicolumn{3}{r}{\text{where } F(g) \ = \ \text{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}[\![S]\!],\ \text{I})} \end{array}$$

For comparison a continuation semantics is also defined.

$$\begin{array}{lcl} \mathcal{S}[\![x := a]\!]\ c\ s & = & c(s[x \mapsto \mathcal{A}[\![a]\!]\ s]) \\ \mathcal{S}[\![\texttt{skip}]\!]\ c\ s & = & c(s) \\ \mathcal{S}[\![S_1;\ S_2]\!] & = & \mathcal{S}[\![S_1]\!] \circ \mathcal{S}[\![S_2]\!] \\ \mathcal{S}[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]\ c & = & \text{cond}(\mathcal{B}[\![b]\!],\ \mathcal{S}[\![S_1]\!]\ c,\ \mathcal{S}[\![S_2]\!]\ c) \\ \mathcal{S}[\![\texttt{while } b \texttt{ do } S]\!] & = & \text{fix}(F) \\ \multicolumn{3}{r}{\text{where } F(g)(c) \ = \ \text{cond}(\mathcal{B}[\![b]\!],\ g \circ \mathcal{S}[\![S]\!]\ g(c),\ c)} \end{array}$$

Using double fixpoint induction we show that the two semantics are equivalent.

THEOREM 12.

$$\mathcal{S}_{cs}[\![S]\!]\ c = c \circ \mathcal{S}_{ds}[\![S]\!]$$

The current proof requires 1110 lines of the .prf file is to show that the needed predicate $P(\phi, \psi) = \forall c : \phi(c) = c \circ \psi$ is admissible (by which we mean that if P is true for all elements of a directed set D then $P(\bigsqcup D)$ is also true. Note carefully that we make no assumption that ψ or c are continuous, they are merely arbitrary partial functions); a mere 368 lines then suffice to perform the induction. It is always possible I'm missing something obvious concerning the admissibility of the predicate I've defined.

By now the urge to prove that a simple compiler was correct had become overwhelming [15]. An obvious first step is to show that the direct semantics and a Structural Operational Semantics were equivalent. (442 and 636 lines of the .prf file were required for each induction.) A bisimulation then shows that this compiler and abstract machine are equivalent. Rounding things out an equivalent natural semantics and a sound and complete axiomatic semantics are also provided.

An obvious next step is to define the computability of a function in terms of a `While` program rather than as a hard-to-read abstract machine-code program. That, however, remains a task for another day.

One problem with the formulation of the denotational semantics is that we have used fixpoints, without showing that they exist (or indeed are unique). We now rectify this omission, and discover the link to continuity.

6.1 Summary

Currently, the library of semantics for `While` has 99 theorems, the proof of which requires 10,252 lines in the .prf files. This library is now ready for release. The source for this material is [18] which I still use as our final year undergraduate text book. Alternative resources are provided in [9, 11].

7. SCOTT TOPOLOGY

If we exclude the pathalogical case of the trivial partial order ($x \leq y$ if, and only if, $x = y$), then the Scott Topology is an example of a non-Hausdorff Topology. This was the initial reason for attempting this material: had I captured non-metricizable topology accurately? Before describing the Scott Topology, we will need to discuss *Directed Complete Partial Orders*. These are partial orders with an additional property.

Definition 11. A non-empty set D is *directed* (under the partial order $\leq$), if for any two elements $x, y \in D$ there exists $z \in D$ with $x \leq z$ and $y \leq z$.

A chain is a special case of directed set in which we restrict z to be one or other of x or y. The partial order $\leq$ is then said to be *Directed Complete* if every directed set has a least upper bound. If, in addition, there is a least element, then we have a *Pointed Directed Complete Partial Order*.

Definition 12. The element y is a member of the *lower set* of the set S if, and only if, there is an element $x \in S$ with $y \leq x$.

Obviously, there is a lower set associated with each element x (consisting of all elements $y \leq x$); we write this as down(x). (Note: although lower_set? is the same as Jerry James' prefix?, my down function is defined over transitive relations while Jerry's almost equivalent function upto insists that the relation be an order relation. As I recall, this subtle distinction is enough to cause a problem. The matter of lower_set? *versus* prefix? is merely one of taste.

We are now ready to define the closed sets of the *Scott Topology*.

Definition 13. A lower set L is a closed set of the Scott Topology, if every directed subset of L has a least upper bound in L.

After a certain amount of effort one determines that this is indeed a topology. Next, we characterize the continuous functions from one Scott Topology to another.

Definition 14. A function $f : T_1 \to T_2$ is *Scott Continuous* whenever

$$\begin{array}{lcl} \forall x, y \in T_1 \,.\, x \sqsubseteq_1 y & \Rightarrow & f(x) \sqsubseteq_2 f(y) \quad \text{and} \\ \forall D \,.\, \text{directed?}(D) & \Rightarrow & f(\bigsqcup_1 D) = \bigsqcup_2 \{f(x) \mid x \in D\} \end{array}$$

($\sqsubseteq_i$ and $\bigsqcup_i$ are respectively the directed complete partial order and least upper bound for T_i, $i = 1, 2$.)

A modicum of effort (733 lines in scott_continuity.prf) suffices to show the following equivalence.

THEOREM 13. *A function $f : T_1 \to T_2$ is (topologically-) continuous if, and only if, it is Scott Continuous.*

Rather more effort (1221 lines in scott_continuity.prf) is required to show that the following innocuous theorem is true.

THEOREM 14. *The pointwise-ordered set of continuous functions $f : T_1 \to T_2$ is directed complete.*

A brief note on this theorem: much time can be wasted trying to restrict the continuous functions so that only computable funtions are permitted [1]. Instead, I propose that we prove properties in the general setting and then deal with the restrictions (*c.f.* constructive analysis and its projections into classical analysis).

If we also insist that the partial order has a least element $\perp$, then we can define fixpoints.

Definition 15. Let $\phi : T \to T$ be an increasing function, then we define the *least fixpoint* of ϕ as

$$\text{fix}(\phi) = \bigsqcup_{n=0}^{\infty} \{\phi^n(\perp)\}$$

If the function being fixpointed is not merely increasing, but also continuous we have in addition:

THEOREM 15. *Let $\psi : T \to T$ be a continuous function, then*

$$\begin{array}{lcl} \psi(\mathit{fix}(\psi)) & = & \mathit{fix}(\psi) \\ \psi(x) \sqsubseteq x & \Rightarrow & \mathit{fix}(\psi) \sqsubseteq x \end{array}$$

The second result is the order-theoretic counterpart to Banach's Contraction Theorem.

7.1 Summary

The library of results for the scott topology has 186 theorems, the proof of which requires 23,269 lines in the .prf files. Although ready for release, ideally one would like to deal with recursively defined Domains and to have dealt with a λ-calculus of some sort [20]. This material is culled from various sources. The driver is the material needed for the denotational semantics in [18]. Other elements are derived from [8, 21] and private conversations with Harold Simmons.

8. CONCLUSIONS

Although I have had to condense the presentation of the work dramatically, hopefully enough of the content survives to give a flavour of what can be achieved using PVS. The primary goal is not to explore the mechanization of these proofs, as by and large the proofs have little in common with one another. Instead, the aim is to provide a solid foundation on top of which such mechanizatio might be built. In

other words, having built a sound foundation for measure theoretic integration, one could then construct automatic tools to perform tasks such as symbolic integration; expecting an automated to tool to build the generalized foundations of integration appears to be a mite optimistic.

I think that my conclusions about using PVS can be captured in two key ideas.

- If you need to extend the real numbers in some way (the extended reals, complex numbers *etc.*), then leave the representation explicit and do *not, under any circumstances,* embed the new number system into the number_field library.
- Always prove the most general results in the most abstract setting available. If you don't, it is inevitable that you will almost surely need to re-prove your results later.

The first point is widely known, but bears repeating: using non-standard number systems confuses the PVS decision procedures sufficiently that it stops using it's built-in rules *even on standard number systems*. The second point is equally well known, but the cost of re-engineering the interface to a released library is much higher than the cost of just doing the job properly in the first place.

I have omitted discussing a library for differentiation in Hilbert Spaces (material derived from [10]), as I had inadvertently restricted the dimension of the space to be countable. In addition I had disobeyed the first rule above and had developed a definition of ∇ (the differential operator) on countably-dimensioned vector spaces of complex numbers. The system was completely unusable, as all of the arithmetic manipulation (especially associativity and commutativity) needed to be manually performed.

Are there any areas of mathematics where PVS is tricky to deploy? Yes: category theory, and it's "types" which mesh badly with the PVS type system. The genuine tension is then between a type system that adds value to proofs of program correctness (say, PVS) and a system sufficiently lax that category theory becomes expressible. For the mathematical sub-topics alluded to in this paper there appear to be few problems. That is to say: making the libraries sufficiently general and usable is seldom a problem, unless one is forced to go beyond the safe ground of the PVS reals in one's number systems. There is also the perennial issue of accurately capturing mathematical concepts in PVS; for topology, I claim that the applications outlined amply demonstrate that the library *is* sufficiently general and useful.

9. ACKNOWLEDGMENTS

Elements of this work have been undertaken at a number of different institutions and with individuals who have been kind enough to host me: NASA Langley (Rick Butler), NIA (César Muñoz), Univesité de Perpignan (Marc Daumas).

My debt to the PVS community should be obvious. I am aware that I have used copious amounts of material provided in the NASA PVS libraries by César Muñoz (vectors), Jerry James (sets_aux), Alfons Geser (orders), and of course Bruno Dutertre and Rick Butler (analysis). If I've omitted to mention you then please introduce yourself at the Workshop. In Manchester, I have benefited from discussions about mathematical topics and perspectives with Harold Simmons, Peter Aczel and Andrea Schalk.

The work outlined in this paper is partially funded by EPSRC grant EP/DO 7908X/1.

10. REFERENCES

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3. Oxford University Press, 1994.

[2] T. Apostol. *Mathematical Analysis*. Addison-Wesley, 2 edition, 1974.

[3] S. Berberian. *Fundamentals of Real Analysis*. Springer-Verlag, 1999.

[4] J. Bialas. σ-additive measure theory. *Journal of Formalized Mathematics*, 2, 1990.

[5] J. Bialas. Properties of caratheodor's measure. *Journal of Formalized Mathematics*, 14, 1992.

[6] N. Bourbaki. *General Topology*. Addison-Wesley, 1966.

[7] N. Cutland. *Computability*. Cambridge University Press, 1980.

[8] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[9] J. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[10] J. Dettman. *Mathematical Methods in Physics and Engineering*. McGraw-Hill, 1962.

[11] M. Gordon. *The Denotational Description of Programming Languages*. Springer-Verlag, 1979.

[12] G. Grimmett and D. Stirzaker. *Probability and Random Processes*. Oxford University Press, 1982.

[13] P. Halmos. *Measure Theory*. van Nostrand, 1950.

[14] J. Hurd. *Formal verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[15] D. Lester. *Combinator Graph Reduction: A Congruence and its Applications*. PhD thesis, Oxford University, 1988.

[16] A. Nedzusiak. σ-fields and probability. *Journal of Formalized Mathematics*, 1, 1989.

[17] A. Nedzusiak. Probability. *Journal of Formalized Mathematics*, 2, 1990.

[18] H. Nielson and F. Nielson. *Semantics with Applications*. John Wiley and Sons, 1992.

[19] J. Shepherdson and H. Sturgis. Computability of recursive functions. *J. ACM*, 10:217–255, 1963.

[20] H. Simmons. *Derivation and Computation*. Cambridge University Press, 2000.

[21] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach*. MIT Press, 1977.

[22] W. Sutherland. *Introduction to Metric and Topological Spaces*. Oxford University Press, 1975.

[23] A. Weir. *Lebesgue Integration and Measure*. Cambridge University Press, 1973.

APPENDIX
A. TOPOLOGICAL PROOF OF THE CONTINUITY OF COMPOSITION

`composition_continuous:`

|
|---|
| {1} $\forall\ (f\colon\ [T_2\ \to\ T_3]\ ,\ g\colon\ [T_1\ \to\ T_2]):$
 $\quad$ continuous?$(g)\ \wedge$ continuous?$(f)\ \Rightarrow$ continuous?$((f \circ g))$ |

Skolemizing and flattening,

`composition_continuous:`

{-1} continuous?(g')
{-2} continuous?(f')
{1} continuous?$((f' \circ g'))$

Rewriting using continuous_open_sets, matching in * (3 times),

`composition_continuous:`

{-1} $\forall\ (\mathrm{Y}\colon\ \mathrm{set}[T_2]):\ \mathrm{open?}[T_2\ ,\ T](\mathrm{Y})\ \Rightarrow\ \mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}(g'\ ,\ \mathrm{Y}))$
{-2} $\forall\ (\mathrm{Y}\colon\ \mathrm{set}[T_3]):\ \mathrm{open?}[T_3\ ,\ U](\mathrm{Y})\ \Rightarrow\ \mathrm{open?}[T_2\ ,\ T](\mathrm{inverse\_image}(f'\ ,\ \mathrm{Y}))$
{1} $\forall\ (Y\colon\ \mathrm{set}[T_3]):$ $\quad \mathrm{open?}[T_3\ ,\ U](Y)\ \Rightarrow\ \mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}((f' \circ g')\ ,\ Y))$

Skolemizing and flattening,

`composition_continuous:`

{-1} $\forall\ (\mathrm{Y}\colon\ \mathrm{set}[T_2]):\ \mathrm{open?}[T_2\ ,\ T](\mathrm{Y})\ \Rightarrow\ \mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}(g'\ ,\ \mathrm{Y}))$
{-2} $\forall\ (\mathrm{Y}\colon\ \mathrm{set}[T_3]):\ \mathrm{open?}[T_3\ ,\ U](\mathrm{Y})\ \Rightarrow\ \mathrm{open?}[T_2\ ,\ T](\mathrm{inverse\_image}(f'\ ,\ \mathrm{Y}))$
{-3} $\mathrm{open?}[T_3\ ,\ U](Y')$
{1} $\mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}((f' \circ g')\ ,\ Y'))$

Instantiating the top quantifier in -2 with the terms: Y', top quantifier in -1 with the terms: inverse_image(f', Y'), then simplifying, rewriting, and recording with decision procedures,

`composition_continuous:`

{-1} $\mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}(g'\ ,\ \mathrm{inverse\_image}(f'\ ,\ Y')))$
{-2} $\mathrm{open?}[T_2\ ,\ T](\mathrm{inverse\_image}(f'\ ,\ Y'))$
{-3} $\mathrm{open?}[T_3\ ,\ U](Y')$
{1} $\mathrm{open?}[T_1\ ,\ S](\mathrm{inverse\_image}((f' \circ g')\ ,\ Y'))$

Expanding the definitions of o, inverse_image, and member,

`composition_continuous:`

{-1} $\mathrm{open?}[T_1\ ,\ S](\{\mathrm{x}\colon\ T_1\ \mid\ Y'(f'(g'(\mathrm{x})))\})$
{-2} $\mathrm{open?}[T_2\ ,\ T](\{x\colon\ T_2\ \mid\ Y'(f'(x))\})$
{-3} $\mathrm{open?}[T_3\ ,\ U](Y')$
{1} $\mathrm{open?}[T_1\ ,\ S](\{\mathrm{x}\colon\ T_1\ \mid\ Y'(f'(g'(\mathrm{x})))\})$

which is trivially true. This completes the proof of `composition_continuous`.

B. METRIC SPACE PROOF OF THE CONTINUITY OF COMPOSITION

`composition_continuous:`

$$\frac{}{\{1\}\quad \forall\ (f\colon\ [T_2\ \to\ T_3],\ g\colon\ [T_1\ \to\ T_2])\colon \\ \quad \text{metric\_continuous?}(g)\ \wedge\ \text{metric\_continuous?}(f)\ \Rightarrow\ \text{metric\_continuous?}((f \circ g))}$$

Expanding the definition of metric_continuous?, metric_continuous_at?, and Skolemizing and flattening,

`composition_continuous:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \forall\ (x\colon\ T_1)\colon \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_1)\colon\ \text{x} \in \text{ball}(x,\ \delta)\ \Rightarrow\ g'(\text{x}) \in \text{ball}(g'(x),\ \varepsilon) \\ \{\text{-}2\} & \forall\ (x\colon\ T_2)\colon \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_2)\colon\ \text{x} \in \text{ball}(x,\ \delta)\ \Rightarrow\ f'(\text{x}) \in \text{ball}(f'(x),\ \varepsilon)\end{array}}{\{1\}\quad \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (x\colon\ T_1)\colon\ x \in \text{ball}(x',\ \delta)\ \Rightarrow\ (f' \circ g')(x)\ \in \text{ball}((f' \circ g')(x'),\ \varepsilon)}$$

Expanding the definition of ball, and member, Skolemizing and flattening,

`composition_continuous:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \forall\ (x\colon\ T_1)\colon \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_1)\colon\ d_1(x,\ \text{x}) < \delta\ \Rightarrow\ d_2(g'(x),\ g'(\text{x})) < \varepsilon \\ \{\text{-}2\} & \forall\ (x\colon\ T_2)\colon \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_2)\colon\ d_2(x,\ \text{x}) < \delta\ \Rightarrow\ d_3(f'(x),\ f'(\text{x})) < \varepsilon\end{array}}{\{1\}\quad \exists\ \delta\colon \forall\ (x\colon\ T_1)\colon\ d_1(x',\ x) < \delta\ \Rightarrow\ d_3((f' \circ g')(x'),\ (f' \circ g')(x)) < \varepsilon'}$$

Instantiating the top quantifier in - with the terms: x', the top quantifier in -2 with the terms: $g'(x')$, and then the top quantifier in -2 with the terms: ε',

`composition_continuous:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_1)\colon\ d_1(x',\ \text{x}) < \delta\ \Rightarrow\ d_2(g'(x'),\ g'(\text{x})) < \varepsilon \\ \{\text{-}2\} & \exists\ \delta\colon \forall\ (\text{x}\colon\ T_2)\colon\ d_2(g'(x'),\ \text{x}) < \delta\ \Rightarrow\ d_3(f'(g'(x')),\ f'(\text{x})) < \varepsilon'\end{array}}{\{1\}\quad \exists\ \delta\colon \forall\ (x\colon\ T_1)\colon\ d_1(x',\ x) < \delta\ \Rightarrow\ d_3((f' \circ g')(x'),\ (f' \circ g')(x)) < \varepsilon'}$$

Skolemizing and flattening,

`composition_continuous:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \forall\ \varepsilon\colon \exists\ \delta\colon \forall\ (\text{x}\colon\ T_1)\colon\ d_1(x',\ \text{x}) < \delta\ \Rightarrow\ d_2(g'(x'),\ g'(\text{x})) < \varepsilon \\ \{\text{-}2\} & \forall\ (\text{x}\colon\ T_2)\colon\ d_2(g'(x'),\ \text{x}) < \delta'\ \Rightarrow\ d_3(f'(g'(x')),\ f'(\text{x})) < \varepsilon'\end{array}}{\{1\}\quad \exists\ \delta\colon \forall\ (x\colon\ T_1)\colon\ d_1(x',\ x) < \delta\ \Rightarrow\ d_3((f' \circ g')(x'),\ (f' \circ g')(x)) < \varepsilon'}$$

Instantiating the top quantifier in - with the terms: δ',

`composition_continuous:`

$$\frac{\begin{array}{ll}\{\text{-}1\} & \exists\ \delta\colon \forall\ (\text{x}\colon\ T_1)\colon\ d_1(x',\ \text{x}) < \delta\ \Rightarrow\ d_2(g'(x'),\ g'(\text{x})) < \delta' \\ \{\text{-}2\} & \forall\ (\text{x}\colon\ T_2)\colon\ d_2(g'(x'),\ \text{x}) < \delta'\ \Rightarrow\ d_3(f'(g'(x')),\ f'(\text{x})) < \varepsilon'\end{array}}{\{1\}\quad \exists\ \delta\colon \forall\ (x\colon\ T_1)\colon\ d_1(x',\ x) < \delta\ \Rightarrow\ d_3((f' \circ g')(x'),\ (f' \circ g')(x)) < \varepsilon'}$$

Skolemizing and flattening,

`composition_continuous`:

{-1}	$\forall$ (x: T_1): $d_1(x', x) < \delta'' \Rightarrow d_2(g'(x'), g'(x)) < \delta'$
{-2}	$\forall$ (x: T_2): $d_2(g'(x'), x) < \delta' \Rightarrow d_3(f'(g'(x')), f'(x)) < \varepsilon'$
{1}	$\exists\ \delta$: $\forall$ (x: T_1): $d_1(x', x) < \delta \Rightarrow d_3((f' \circ g')(x'), (f' \circ g')(x)) < \varepsilon'$

Instantiating the top quantifier in + with the terms: δ'',

`composition_continuous`:

{-1}	$\forall$ (x: T_1): $d_1(x', x) < \delta'' \Rightarrow d_2(g'(x'), g'(x)) < \delta'$
{-2}	$\forall$ (x: T_2): $d_2(g'(x'), x) < \delta' \Rightarrow d_3(f'(g'(x')), f'(x)) < \varepsilon'$
{1}	$\forall$ (x: T_1): $d_1(x', x) < \delta'' \Rightarrow d_3((f' \circ g')(x'), (f' \circ g')(x)) < \varepsilon'$

Skolemizing and flattening,

`composition_continuous`:

{-1}	$\forall$ (x: T_1): $d_1(x', x) < \delta'' \Rightarrow d_2(g'(x'), g'(x)) < \delta'$
{-2}	$\forall$ (x: T_2): $d_2(g'(x'), x) < \delta' \Rightarrow d_3(f'(g'(x')), f'(x)) < \varepsilon'$
{-3}	$d_1(x', x'') < \delta''$
{1}	$d_3((f' \circ g')(x'), (f' \circ g')(x'')) < \varepsilon'$

Instantiating the top quantifier in - with the terms: x'', and then the top quantifier in - with the terms: $g'(x'')$,

`composition_continuous`:

{-1}	$d_1(x', x'') < \delta'' \Rightarrow d_2(g'(x'), g'(x'')) < \delta'$
{-2}	$d_2(g'(x'), g'(x'')) < \delta' \Rightarrow d_3(f'(g'(x')), f'(g'(x''))) < \varepsilon'$
{-3}	$d_1(x', x'') < \delta''$
{1}	$d_3((f' \circ g')(x'), (f' \circ g')(x'')) < \varepsilon'$

Expanding the definition of o,

`composition_continuous`:

{-1}	$d_1(x', x'') < \delta'' \Rightarrow d_2(g'(x'), g'(x'')) < \delta'$
{-2}	$d_2(g'(x'), g'(x'')) < \delta' \Rightarrow d_3(f'(g'(x')), f'(g'(x''))) < \varepsilon'$
{-3}	$d_1(x', x'') < \delta''$
{1}	$d_3(f'(g'(x')), f'(g'(x''))) < \varepsilon'$

Simplifying, rewriting, and recording with decision procedures,

This completes the proof of `composition_continuous`.

Model Checking for the Practical Verificationist:

A User's Perspective on SAL

Lee Pike
Galois, Inc.
leepike@galois.com

ABSTRACT

SRI's Symbolic Analysis Laboratory (SAL) is a high-level language-interface to a collection of state-of-the-art model checking tools. SAL contains novel and powerful features, many of which are not available in other model checkers. In this experience report, I highlight some of the features I have particularly found useful, drawing examples from published verifications using SAL. In particular, I discuss the use of higher-order functions in model checking, infinite-state bounded model checking, compositional specification and verification, and finally, mechanical theorem prover and model checker interplay. The purpose of this report is to expose these features to working verificationists and to demonstrate how to exploit them effectively.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: [formal methods, model checking]

1. INTRODUCTION

SRI's Symbolic Analysis Laboratory (SAL)[1] is bad news for interactive mechanical theorem provers. SAL is so automated yet expressive that for many of the verification endeavors I might have previously used a mechanical theorem prover, I would now use SAL. The purpose of this brief report is to persuade you to do the same.

To convince the reader, I highlight SAL's features that are especially useful or novel from a practitioner's perspective. My goals in doing so are (1) to begin a dialogue with other SAL users regarding how best to exploit the tool, (2) to show off SAL's features to verificationists not yet using SAL, and (3) to provide user feedback to spur innovation in the dimensions I have found most novel and beneficial, as a user.

[1]SAL is open source under a GPL license and the tool, documentation, a user-community wiki, etc. are all available at `http://sal.csl.sri.com`.

AFM'07, November 6, Atlanta, GA, USA.

With my coauthors, I have had the opportunity to use SAL in a number of applied verifications [3, 4, 5, 20, 21, 22].[2] These works draw from the domains of distributed systems, fault-tolerant protocols, and asynchronous hardware protocols (the most notable omission is the domain of software, although the techniques reported are not domain-specific).

Let me also say what are *not* the intentions of this report. This report is not a manual or user's guide; such artifacts are available from the SAL website (I do, however, strive to make the report self-contained, even for the reader not experienced with SAL). Also, I do not compare and contrast SAL to other model checkers. In particular, I am not claiming that each feature highlighted is unique to SAL, but I do claim that their combination in one tool is unique.

The outline for the remainder of this report is as follows. In Section 2, I briefly overview SAL's language and toolset. In Section 3, I describe how higher-order functions, as implemented in SAL, can be used in a model checking context to simplify the specification of state machines. In Section 4, I describe SAL's infinite-state bounded model checker, a particularly novel and powerful model checker, and I describe how it can be used to prove safety-properties of real-time systems with an order of magnitude reduction in proof steps as compared to mechanical theorem proving. Section 5 describes how to judiciously use synchronous and asynchronous composition to ease the proof effort required in infinite-state bounded model checking and to decompose environmental constraints from system specifications. Despite the power of SAL, a model checker sometimes is not enough; in Section 6, I describe cases in which SAL can be effectively used in tandem with a mechanical theorem prover. Conclusions are provided in Section 7.

2. SAL OVERVIEW

SAL has a high-level modeling language for specifying state machines. A state machine is specified by a *module*. A module consists of a set of state variables (declared to be *input*, *output*, *global*, or *local*) and guarded transitions. A guarded transitions is *enabled* if its guard—some expression that evaluates to a boolean value—is true. Of the enabled transitions in a state, one is nondeterministically executed. When a transition is exercised, the next-state values are

[2]My coathors for these works include Geoffrey Brown, Steve Johnson, Paul Miner, and Wilfredo Torres-Pomales. The specifications associated with these works are all available from `http://www.cs.indiana.edu/~lepike`.

assigned to variables; for example, consider the following guarded transition:

```
H --> a' = a - 1;
      b' = a;
      c' = b' + 1;
```

If the guard `H` holds and the transition is exercised, then in the next state, the variable `a` is decremented by one, the variable `b` is updated to the previous value of `a`, and the variable `c` is updated to the new value of `b`, plus one. In the language of SAL, "`;`" denotes statement separation, not sequential composition (thus, variable assignments can be written in any order). If no variables are updated in a transition (i.e., `H -->`), the state idles.

Modules can be composed both synchronously (`||`) and asynchronously (`[]`), and composed modules communicate via shared variables. In a synchronous composition, a transition from each module is simultaneously applied; a synchronous composition is deadlocked if either module has no enabled transition. Furthermore, a syntactic constraint on modules requires that no two modules update the same variable in a synchronous composition. In an asynchronous composition, an enabled transition from exactly one of the modules is nondeterministically applied.

The language is typed, and predicate sub-types can be declared. Types can be both interpreted and uninterpreted, and base types include the reals, naturals, and booleans. Array types, inductive data-types, and tuple types can be defined. Both interpreted and uninterpreted constants and functions can be specified.

One of the greatest practical benefits of SAL is that a variety of useful tools are associated with the same input language. SAL 3.0 includes a BDD-based model checker, a SAT based model checker (capable of performing k-induction proofs), and infinite-state bounded model checker that is integrated with the Yices satisfiability modulo theories (SMT) solver, a BDD-based simulator, a BDD-based deadlock checker, and an automated test-case generator. Other tools can be built on SAL's API.

3. HIGHER-ORDER FUNCTIONS

The first feature of SAL I cover is higher-order functions. What use are higher-order functions in a model checker? Model checkers are about building state machines, and higher-order functions are typically associated with "stateless" programming. The practicality of higher-order functions is well-known in the programming and mechanical theorem proving communities [12], and these advantages apply just as well to specifying the functional aspects of a model. Furthermore, higher-order functions are just as indispensable for specifying a model's stateful aspects; I give two supporting examples below. First, I show how sets can be specified with higher-order functions, allowing the easy specification of nondeterministic systems. Second, I show how to replace guarded transitions with higher-order functions; the benefit of doing so is that it allows one to decompose the environment and system specifications or to make assumptions explicit in proofs.

3.1 Sets and Nondeterminism

The first example is drawn from work done with Geoffrey Brown to verify real-time physical-layer protocols [3, 5]. Suppose I want to nondeterministically update some value to be within a parameterized closed interval of real-time (modeled by the real number line). We can define a higher-order function that takes a minimum and a maximum value and returns the set of real values in the closed interval between them. The function has the return type of `REAL -> BOOLEAN`, which is the type of a set of real numbers.

```
timeout( min : REAL
       , max : REAL) : [REAL -> BOOLEAN] =
  {x : REAL |     min <= x
              AND x <= max};
```

I use the identifier '`timeout`' to pay homage to *timeout automata*, a theory and corresponding implementation in SAL developed by Bruno Dutertre and Maria Sorea for specifying and verifying real-time systems using infinite-state k-induction (see Section 4) [10].

Then, in specifying the guarded transitions in a state machine, we can simply call the function with specific values (let `H` and `I` be some predicates on the real-time argument). The `IN` operator specifies that its first argument, a variable of some arbitrary type `T`, nondeterministically takes a value in its second argument, a set of type `T -> BOOLEAN`.

```
   H(t) --> t' IN timeout(1, 2);
[] I(t) --> t' IN timeout(3, 5);
```

With higher-order functions, nondeterministic transitions can be specified succinctly, as above, rather than specifying an interval in each transition.[3]

3.2 Specifying Transitions

Another benefit of higher-order functions is that they can be used to "pull" constraints out of the state machine specification. The motivations for doing this include (1) to simplify specifications, (2) to make assumptions and constraints more explicit in proofs, and (3) to decompose environmental constraints from state machine behavior.

To illustrate this idea, we will model check a simple distributed system built from a set of nodes (of the uninterpreted type `NODE`) and a set of one-way channels between nodes. Furthermore, suppose that the creation of channels is dynamic, and we wish to make this explicit in our state-machine model of the system. We might prove properties like, "If there is a channel pointing from node `n` to node `m`, then there is no channel pointing from `m` to `n`," (i.e., channels are unidirectional) or "No channel points from a node in one subset of nodes to a node in another subset of nodes."

[3]In this and subsequent specifications, we sometimes state just the guarded transitions where the remainder of the guard specification is irrelevant or can be inferred from context.

To build the model, we will record the existence of channels using a `NODE` by `NODE` matrix of booleans. For convenience, we define `CHANS` to be the type of these matrices:

```
CHANS : TYPE = ARRAY NODE OF ARRAY NODE OF BOOLEAN;
```

For some matrix `chans` of type `CHANS`, we choose, by convention, to let `chans[a][b]` mean that there is a channel pointing from node `a` to node `b`.

We define a function `newChan` that takes two nodes and adds a channel between them. Indeed, having higher-order functions at our disposal, we define the function in curried form.

```
newChan(a : NODE, b : NODE) : [CHANS -> CHANS] =
  LAMBDA (chans : CHANS) :
    chans WITH [a][b] := TRUE;
```

The function `newChan` returns a function, and that function takes a set of channels and updates it with a channel from `a` to `b`.

Now we will build a state machine containing three asynchronous transitions. Two of the transitions introduce new channels into the system depending on the current system state, and the final one simply stutters (i.e., maintains the system state). Let `H` and `I` be predicates over sets of channels (i.e., functions of type `CHANS -> BOOLEAN`), and let `m`, `n`, `o`, `p`, ... be constant node-identifiers (i.e., constants of type `NODE`).

```
   H(chans) --> chans' =  newChan(n, m)
                          (newChan(p, q)
                          (chans));
[] I(chans) --> chans' =  newChan(q, n)(chans);
[] ELSE -->
```

Alternatively, we can write an equivalent specification using a predicate rather than defining three transitions in the state machine. First, we define a function that takes a current channel configuration and returns a set of channel configurations—i.e., `chanSet` returns a predicate parameterized by its argument.

```
chanSet(chans : CHANS) : [CHANS -> BOOLEAN] =
  {x : CHANS |
         (H(chans) => x =  newChan(n, m)
                           (newChan(p, q)
                           (chans)))
     AND (I(chans) => x =  newChan(q, n)(chans))
     AND (   (NOT I(chans) AND NOT H(chans))
          => FORALL (a, b : NODE) :
               x[a][b] = chans[a][b]))};
```

Now, we can specify the state-machine with the following single transition:

```
TRUE --> chans' IN chanSet(chans);
```

In all states, `chans` is updated to be some configuration of channels from the possible configurations returned by `chanSet(chans)`. Why might one wish to "pull" transitions out of a state-machine specification and into a predicate? One reason would be to decompose the transitions enabled by the state machine itself from the environmental constraints over the machine. For example, suppose that in our example distributed system, the nodes themselves are not responsible for creating new channels—some third party does so. In this case, specifying channel creation in the transitions of the state-machine module itself belies the distinction between the distributed system and its environment. (See Section 5.2 for an industrial application of this idea.)

We can even go one step further and remove the constraints entirely from the model. First, we modify the previous transition so that it is completely unconstrained: under any condition (i.e., a guard of `TRUE`), it returns any configuration of channels. (We also add an auxiliary history variable, the sole purpose of which is to record the set of channels in the previous state; we use the variable shortly.

```
TRUE --> chans' IN {x : CHANS | TRUE};
         chansHist' = chans
```

Now suppose we were to prove some property about the machine; for instance, suppose we wish to prove that all channels between two nodes are unidirectional. We might state the theorem as follows.[4]

```
Thm : THEOREM system |-
  G(FORALL (a,b : NODE) :
      chans[a][b] => NOT chans[b][a]);
```

With a completely under-specified state machine, the theorem fails. We are forced to add as an explicit hypothesis that the state variable `chans` belongs to the set of channels `chanSet(chansHist)` generated in the previous state.[5]

```
Thm : THEOREM system |-
  LET inv : BOOLEAN = chanSet(chansHist)(chans)
  IN W( (inv => FORALL (a, b : NODE) :
                  chans[a][b] => NOT chans[b][a])
       , NOT inv);
```

[4]This and subsequent SAL theorems, lemmas, etc. are stated in the language of Linear Temporal Logic (LTL), a common model-checking logic. In the following theorem, the `G` operator states that its argument holds in all states on an arbitrary path, and LTL formulas are implicitly quantified over all paths in the system. See the SAL documentation for more information.

[5]Due to the semantics of LTL, we cannot simply add `chanSet(chansHist)(chans)` (call it `inv`) as a hypothesis. This is because false positives propagate over transitions: there is a path on which `inv` fails for one or more steps (so the implication holds), and because the state machine was under-specified, we can then reach a state in which `inv` holds but the unidirectional property fails. To solve this problem, we use the *weak until* LTL operator `W`. Intuitively, `W` states that on any arbitrary path, its first argument either holds forever, or it holds in all states until its second argument holds, at which point neither need to hold.

Being forced to add the hypothesis can be a virtue: the environmental assumptions now appear in the theorem rather than being implicit in the state machine. The difference between assumptions being implicit in the model or explicit in the theorem is analogous to postulating assumptions as axioms in a theory or as hypotheses in a proof—a classic tradeoff made in mechanical theorem proving. SAL allows a verificationist to have the same freedoms in a model checking environment.

4. PRACTICAL INVARIANTS

Bounded model checkers have historically been used to find counterexamples, but they can also be used to prove invariants by induction over the state space [7]. SAL supports *k-induction*, a generalization of the induction principle, which can prove some invariants that may not be strictly inductive. The technique can be applied to both finite-state and infinite-state systems. In both cases, a bounded model checker is used. For infinite-state systems, the bounded model checker is combined with a *satisfiability modulo theories* (SMT) solver [8, 26]. For shorthand, I refer to infinite-state bounded model checking via k-induction as *inf-bmc* in the remainder of this paper.

The default SMT solver used by SAL is SRI's own Yices solver, which is a SMT solver for the satisfiability of (possibly quantified) formulas containing uninterpreted functions, real and integer linear arithmetic, arrays, fixed-size bit vectors, recursive datatypes, tuples, records, and lambda expressions [9]. Yices has regularly been one of the fastest and most powerful solvers at the annual SMT competitions [2].

4.1 Generalized Induction

To define k-induction, let $(S, I, \rightarrow)$ be a transition system where S is a set of states, $I \subseteq S$ is a set of initial states, and $\rightarrow$ is a binary transition relation. If k is a natural number, then a *k-trajectory* is a sequence of states $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ (a 0-trajectory is a single state). Let k be a natural number, and let P be property. The k-induction principle is then defined as follows:

- *Base Case*: Show that for each k-trajectory $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$ such that $s_0 \in I$, $P(s_j)$ holds, for $0 \leq j < k$.
- *Induction Step*: Show that for all k-trajectories $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_k$, if $P(s_j)$ holds for $0 \leq j < k$, then $P(s_k)$ holds.

The principle is equivalent to the usual transition-system induction principle when $k = 1$. In SAL, the user specifies the depth at which to attempt an induction proof, but the attempt itself is automated.

For example, consider the following state machine defined in SAL:

```
counter1 : MODULE =
BEGIN
  OUTPUT cnt : INTEGER
  OUTPUT b : BOOLEAN
INITIALIZATION
  cnt = 0;
  b = TRUE;
TRANSITION
[   b    --> cnt' = cnt + 2;
             b' = NOT b
 [] ELSE --> cnt' = cnt - 1;
             b' = NOT b
]
END;
```

The module produces an infinite sequence of integers and boolean values. It's behavior is as follows:

cnt	:	0	2	1	3	2	4	...
b	:	*T*	*F*	*T*	*F*	*T*	*F*	...

Now suppose we wish to prove the following invariant holds:

```
Cnt1Clm : CLAIM counter1 |- G(cnt >= 0);
```

While `Cnt1Clm` is an invariant, it is not inductive (i.e., $k = 1$). To see why, consider the induction step, and consider (an unreachable) state in which `b` is false and `cnt` is zero. This state satisfies `Cnt1Clm`, but in one step, `cnt` equals -1, and the invariant fails. However, in any two steps (i.e., $k = 2$), the claim holds.

4.2 Disjunctive Invariants

Unfortunately, k-induction is exponential in the size of k, so at some point, an invariant will likely need to be manually strengthened. I find the method of building up invariants using *disjunctive invariants* [24] to be particularly suited to SAL. A disjunctive invariant is built up by adding disjuncts, each of which is an invariant for some system configuration. In SAL, disjunctive invariants can quickly be built up by examining the counterexamples returned by SAL in failed proof attempts. Disjunctive invariants contrast with the traditional approach of strengthening an invariant by adding *conjuncts*. Each conjunct in a traditional invariant needs to hold in every system configuration, unlike in a disjunctive invariant.

Consider the following simple example:

```
counter2 : MODULE =
BEGIN
  OUTPUT cnt : INTEGER
  OUTPUT b : BOOLEAN
INITIALIZATION
  cnt = 0;
  b = TRUE;
TRANSITION
[   b    --> cnt' = (-1 * cnt) - 1;
             b' = NOT b
 [] ELSE --> cnt' = (-1 * cnt) + 1;
             b' = NOT b
]
END;
```

The module produces an infinite sequence of integers and boolean values. It's behavior is as follows:

cnt	:	0	−1	2	−3	4	−5	...
b	:	*T*	*F*	*T*	*F*	*T*	*F*	...

Suppose we wished to prove an invariant that captures the behavior of the state machine. Rather than consider every configuration of the machine, we might begin with an initial approximation that only characterizes states in which `b` is true:

```
Cnt2Clm : CLAIM counter2 |- G(b AND cnt >= 0);
```

The conjecture fails. SAL automatically returns a counterexample in which both `b` is false and `cnt` is less than zero. Guided by the counterexample, we can now augment the original conjecture with a disjunct to characterize the configuration in which `b` is false.

```
Cnt2Clm : CLAIM counter2 |-
  G(   (b AND cnt >= 0)
    OR (NOT b AND cnt < 0));
```

The stated conjecture is proved by SAL. Of course, the example presented is quite simple, but the technique allows one to build up invariants of complex specifications in piecemeal fashion by considering only one configuration at a time and allowing SAL's counterexamples to show remaining configurations.

Using k-induction and disjunctive invariants, Geoffrey Brown and I were able to dramatically reduce the verification effort of physical-layer protocols, such as the Biphase Mark protocol (used in CD-player decoders, Ethernet, and Tokenring) and the 8N1 protocol (used in UARTs) [5]. The verification of BMP presented herein results in an orders-of-magnitude reduction in effort as compared to the protocol's previous formal verifications using mechanical theorem proving. Our verification required 3 invariants, whereas a published proof using the mechanical theorem prover PVS required 37 [28]. Using infinite-bmc induction, proofs of the 3 invariants were completely automated, whereas the PVS proof initially required some 4000 user-supplied proof directives, in total. Another proof using PVS is so large that the tool required 5 hours just to *check* the manually-generated proof whereas the SAL proof is generated automatically in seconds [13]. BMP has also been verified by J. Moore using Nqthm, a precursor to ACL2, requiring a substantial proof effort (Moore cites the work as being one of the "best ideas" of his career) [18].[6] Geoffrey and I spent only a couple of days to obtain our initial results; much more time was spent generalizing the model and writing up the results.

In Section 5.1, we describe techniques to exploit k-induction effectively.

[6] `http://www.cs.utexas.edu/users/moore/best-ideas/`.

5. COMPOSITION

Similar to SMV [16] and other model checkers, SAL allows state machines to be composed both synchronously and asynchronously, and a composed state machine may contain both synchronously and asynchronously composed subcompositions. For example, supposing that `A`, `B`, `C`, and `D` were modules, the following is a valid composition (assuming the modules satisfy the variable-update constraints for synchronous composition mentioned in Section 2):

```
E : MODULE = (A [] B) [] (C || D);
```

The judicious use of synchronous composition can simplify specifications and ease the verification effort, taking further advantage of SAL's tools.

In the following, I first provide an example emphasizing how to use synchronous composition to reduce the proof effort for k-induction. The second emphasizes how to judiciously use composition to decompose environmental constraints from the state machine itself, allowing for simple specification refinements.

5.1 Cheap Induction Proofs

In proofs by k-induction (described in Section 4), k specifies the length of trajectories considered in the base case and induction step. With longer trajectories, weaker invariants can be proved. Unfortunately, the cost of k induction is exponential in the value of k, since a SAT-solver is used to unroll the transition relation. Thus, models that reduce unessential interleavings make k-induction proofs faster.

Let me give a simple example explaining this technique first and then describe how I used it in an industrial verification. Recall the module `counter1` from Section 4.1. We will modify its transitions slightly so that it deadlocks when the counter is greater than 2 (the sole purpose of which is to avoid dealing with fairness conditions in the foregoing state-machine composition):

```
   b AND cnt <= 2       --> cnt' = cnt + 2;
                            b' = NOT b
[] (NOT b) AND cnt <= 2 --> cnt' = cnt - 1;
                            b' = NOT b
```

The behavior of the generated state machine is as follows:

cnt	:	0	2	1	3	(deadlock)
b	:	*T*	*F*	*T*	*F*	(deadlock)

Now suppose we wish to prove that the `cnt` variable is always nonnegative.

```
cntThm : THEOREM nodes |- G(cnt >= 0);
```

This property is k-inductive for $k = 2$ (for the reasons mentioned in Section 4.1). Now we are going to compose some instances of the `node` module together. SAL provides some

convenient syntax for doing this, but first, we must parameterize the above module. We being by defining an index type `[1..I]` denoting the sequence 1, 2, ..., I.

```
I : NATURAL = 5;
INDICES : TYPE = [1..I];
```

Now we modify the declaration of the `node` module from

```
node : MODULE =
BEGIN
  ...
```

as above to a module parameterized by indices:

```
node[i: INDICES]: MODULE =
BEGIN
  ...
```

where the remainder of the module declaration is exactly as presented above. Now we can automatically compose instances of the module together with the following SAL declaration (an explanation of the syntax follows):

```
nodes : MODULE =
  WITH OUTPUT cnts : ARRAY INDICES OF INTEGER
    (|| (i : INDICES) : RENAME cnt TO cnts[i]
                        IN node[i]);
```

The module `nodes` is the synchronous composition of instances of the `node` module, one for each index in `INDICES`. The `nodes` module has only one state variable, `cnts`. This variable is an array, and its values are taken from the `cnt` variables in each module. Thus, `cnts[j]` is value of `cnt` from the jth node module.

We can modify slightly the theorem proved about a single module to cover all of the modules in the composition:

```
cntsThm : CLAIM nodes |-
  G(FORALL (i : INDICES) : cnts[i] >= 0);
```

The theorem is proved using k-induction at $k = 2$. Indeed, in the synchronous composition, it is proved for $k = 2$ for any number of nodes (i.e., values of `I`). I proved `cntsThm` for two through thirty nodes on a PowerBook G4 Mac with one gigabyte of memory, and the proofs all took about 1-2 seconds.

On the other hand, if we change the synchronous composition in the `nodes` module above to an asynchronous composition,[7] the cost increases exponentially. Intuitively, we have to increase the value of k to account for the possible interleavings in which each module's `cnt` and `b` variables are updated. For two nodes (`I` = 2), proving the theorem `cntsThm` requires at minimum $k = 6$. On the same PowerBook, the proof takes about one second. For `I` = 3, the proof requires $k = 10$, and the proof takes about three and one-half seconds. For `I` = 4, we require $k = 14$ and the proof takes just over 10 minutes; for `I` = 5, we require $k = 18$, and I stopped running the experiment after five hours of computation!

This technique has a practical aspect. To verify a reintegration protocol in SAL using inf-bmc, I used these techniques [21]. A *reintegration protocol* is a protocol designed for fault-tolerant distributed systems—in particular, I verified a reintegration protocol for SPIDER, a time-triggered fly-by-wire communications bus being designed at NASA Langley [27]. The protocol increases system survivability by allowing a node that has suffered a transient fault (i.e., it is not permanently damaged) to regain state consistent with the non-faulty nodes and reintegrate with them to deliver system services.

In the model of the system, I initially began with a highly-asynchronous model. I realized, however, that much of the asynchronous behavior could be synchronous without affecting the fidelity of the model. One example is the specification of non-faulty nodes (or *operational nodes*) being observed by the reintegrating node. In this model, their executions are independent of each other, and their order of execution is not relevant to the verification (we do not care which operational node executes first, second, etc., but only that each operational node executes within some window). Thus, we can update the state variables of each operational node synchronously. Each maintains a variable ranging over the reals (its timeout variable—see Section 3.1) denoting at what real-time it is modeled to execute, but their transitions occur synchronously.

An anonymous reviewer of this report noted that the technique appears to be akin to a partial-order reduction [6] applied to inf-bmc, and asked if such a reduction could be realized automatically. For the simple example presented, I believe it would be possible to do so, but as far as I know, generalizations would be an open research question.

To summarize, while synchronous and asynchronous composition are not unique to SAL, their impact on k-induction proofs is a more recent issue. Since k-induction is especially sensitive to the lengths of trajectories to prove an invariant, synchronous composition should be employed when possible.

5.2 Environmental Decomposition

Another use of synchronous composition is to decompose the environment model from the system model. The purpose of an environmental model is to constrain the behavior of a system situated in that environment. In the synchronous composition of modules `A` and `B`, if either module deadlocks, the composition `A || B` deadlocks. Thus, environmental constraints can be modeled by having the environment deadlock the entire system on execution paths outside of the environmental constraints.

[7] The theorem `cntsThm` would have had to include fairness constraints if the asynchronously-composed modules did not deadlock after some number of steps.

For example, Geoffrey Brown and I used this approach in the verification and refinement of physical-layer protocols [3]. Physical-layer protocols are cooperative protocols between a transmitter and a receiver. The transmitter and receiver are each hardware devices driven by separate clocks. The goal of the protocols is to pass a stream of bits from the transmitter to the receiver. The signal must encode not only the bits to be passed but the transmitter's clock signal so that the receiver can synchronize with the transmitter to the extent necessary to capture the bits without error.

In the model, we specify three modules: a transmitter (`rx`), a receiver (`tx`), and a constraint module (`constraint`) simulating the environment. The entire system is defined as the composition of three modules, where the transmitter and receiver are asynchronously composed, and the constraint module is synchronously composed with the entire system:

```
system : MODULE = (tx [] rx);
systemEnv : MODULE = system || constraint;
```

In this model, the constraint module separates out from the system the effects of *metastability*, a phenomenon in which a flip flop (i.e., latch) does not settle to a stable value (i.e., "1" or "0") within a clock cycle. Metastability can arise when a signal is asynchronous (in the hardware-domain sense of the word); that is, it passes between clock regions. One goal of physical-layer protocols is to ensure that the probability of metastable behavior is sufficiently low.

In the module `rx`, the receiver's behavior is under-specified. In particular, we do not constrain the conditions under which metastability may occur. The receiver captures the signal sent with the boolean variable `rbit`. The receiver is specified with a guarded transition like the following (the guard has been elided) allowing `rbit` to nondeterministically take a value of `FALSE` or `TRUE`, regardless of the signal sent to it by the transmitter:

```
... -->  rbit' IN {FALSE, TRUE};
```

The under-specified receiver is constrained by its environment. The constraint module definition is presented below, with extraneous details (for the purposes of this discussion) elided.

```
constraint : MODULE =
    ...
DEFINITION
  stable = NOT changing OR tclk - rclk < TSTABLE;
    ...
TRANSITION
   rclk' /= rclk AND (stable => rbit' = tdata) -->
[] ...
```

In the module, we define the value of the state variable `stable` to be a fixed function of other state variables (modeling the relationship between the transmitter's and receiver's respective clocks). The variable `stable` captures the sufficient constraints to prevent metastability. We give a representative transition in the constraint module. The transition's guard is a conjunction. The first condition holds if the receiver is making a transition (the guard states that the reciever's clock is being updated—this is a timeout automata model, as mentioned in Section 3.1). The second conjunct enforces a relation between the signal the transmitter sends (`tdata`) and the value captured by the receiver (`rbit`): if `stable` holds, then the receiver captures the signal. In other words, the constraint module prunes the execution paths allowed by the `system` module alone in which the value of `rbit` is completely nondeterministic. Finally, note that because no state variables follow `-->` in the transition, no state variables are updated by the environment.

So what are the benefits of the decomposition? One example is refinement. Brown and I wished to refine the physical-layer protocols we specified. These protocols are real-time protocols. Unfortunately, we could not easily compose the real-time specifications with synchronous (i.e., finite-state) hardware specifications of the transmitter and the receiver. Doing so would require augmenting the invariant about real-time behavior with invariants about the synchronous hardware. Ideally, we could decompose the correctness proof of the protocol with the correctness proofs of the hardware in the transmitter and receiver, respectively.

Therefore, we developed a more abstract finite-state, discrete-time model of the protocols. The finite-state model could be easily composed with the other finite-state specifications of the synchronous hardware within a single clock domain; i.e., the transmitter's encoder could be composed with a specification of the remainder of the transmitter's hardware, and the receiver's decoder could be composed with a specification of the remainder of the receiver's hardware. The entire specification could then be verified using a conventional model checker, like SAL's BDD-based model checker.

The goal was to carry out a *temporal refinement* to prove that the real-time implementation refined the discrete-time specification. Using inf-bmc, we verified the necessary refinement conditions. These conditions demonstrate that the implementation is more constrained (i.e., has fewer behaviors) than its specification along the lines of Abadi and Lamport's classic refinement approach [1].

In SAL, most of the refinement was "for free." For example, recall that a synchronous composition `A || B` constrains the possible behaviors of module `A` and `B`. Thus, by definition, `A || B` is a refinement of `A`.

Thus, to prove that the real-time model

```
system : MODULE = (tx [] rx);
systemEnv : MODULE = (tx [] rx) || constraint;
```

refines the discrete-time model

```
system_dt : MODULE = tx_dt [] rx_dt;
```

(where `dt` stands for "discrete time"), we simply had to prove that `tx` refines `tx_dt` and that `rx` refines `rx_dt`. We did not have to refine the `constraint` module, as one would intuitively expect, since it is orthogonal to the system itself.

6. THE MARRIAGE OF MODEL CHECKING AND THEOREM PROVING

Sometimes, even the powerful tools provided by SAL are not enough. In this section, I describe three ways in which I have used SAL in tandem with a mechanical theorem prover to take advantage of the best of both worlds. The two examples include using SAL to discover counterexamples to failed proof conjectures and verifying a theory of real-time systems in a mechanical theorem prover, and then using SAL to prove that an implementation satisfies constraints from the theory.

6.1 Counterexample Discovery

Sometimes verifications require the full interactive reasoning-power of mechanical theorem proving. This is the case when, for example, the specification or proof involves intricate mathematics (that does not fall within a decidable theory), or the specification is heavily parameterized (e.g., proving a distributed protocol correct for an arbitrary number of nodes).

Although rarely discussed in the literature, most attempts to prove conjectures using interactive mechanical theorem proving fail. Only after several iterations of the following steps

1. attempting to prove a theorem,
2. then discovering the theorem is false or the proof is too difficult,
3. revising the specification or theorem accordingly,
4. and repeating from Step 1

is a theorem finally proved.

Provided the theorem prover is sound and the conjecture is not both true and unprovable—a possibility in mathematics—there are two possible reasons for a failed proof attempt. First, the conjecture may be true, but the user lacks the resources or insight to prove it. Second, the conjecture may be false. It can be difficult to determine which of these is the case.

Proofs of correctness of algorithms and protocols often involve nested case-analysis. A proof obligation that cannot be completed is often deep within the proof, where intuition about the system behavior—and what constitutes a counterexample—wanes. The difficulty is also due to the nature of mechanical theorem proving. The proof steps issued in such a system are fine-grained. Formal specifications make explicit much of the detail that is suppressed in informal models. The detail and formality of the specification and proof makes the discovery of a counterexample more difficult.

Paul Miner, Wilfredo Torres-Pomales, and I ran against this very problem when trying to verify the correctness of a fault-tolerant protocol for a distributed real-time fault-tolerant bus [22]. The protocol we were verifying was an interactive consistency protocol designed for NASA's SPIDER architecture [27]. We were verifying the protocol in PVS.

```
[-1]  good?(r_status!1(r!1))
[-2]  asymmetric?(b_status!1(G!1))
[-3]  IC_DMFA(b_status!1, r_status!1, F!1)
[-4]  all_correct_accs?(b_status!1, r_status!1, F!1)
  |-------
[1]   trusted?(F!1`BR(r!1)(G!1))
[2]   declared?(F!1`BB(b2!1)(G!1))
{3}   (FORALL (p_1: below(R)):
         (trusted?(F!1`RB(b1!1)(p_1)) =>
            NOT asymmetric?(r_status!1(p_1))))
       &
       (FORALL (p_1: below(R)):
          (trusted?(F!1`RB(b2!1)(p_1)) =>
             NOT asymmetric?(r_status!1(p_1))))
[4]   declared?(F!1`BB(b1!1)(G!1))
[5]   robus_ic(b_status!1, r_status!1,
                F!1`BB(b1!1)(G!1), F!1`RB(b1!1))
               (G!1, msg!1, b1!1)
       =
       robus_ic(b_status!1, r_status!1,
                 F!1`BB(b2!1)(G!1), F!1`RB(b2!1))
                (G!1, msg!1, b2!1)
```

Figure 1: Unproved PVS Sequent

The protocol suffered a bug in its design: the bug occurrs if two Byzantine faults [15] (allowing unconstrained faulty behavior) occur simultaneously. Such an occurrence is a rare pathological case that escaped our pencil-and-paper analysis.

During the course of formally verifying the protocol, Torres-Pomales independently discovered the bug through "engineering insight." Nevertheless, as a case-study in distilling counterexamples from a failed proof, we decided to press on in the proof until a single leaf in the proof tree remained. To give the reader an idea about what the unproven leaf looked like, we present the PVS sequent if Figure 1 (it is described in detail elsewhere [22]).

The unproven leaf, however, does not give a good idea as to whether a counterexample actually exists and if one does, what that counterexample is. Therefore, building on the specification and verification of a similar protocol done by John Rushby in SAL [25], we formulated the unproven leaf as an LTL conjecture in SAL (Figure 2) stating that in all states, the unproven leaf from the PVS specification indeed holds.

Note the similarity between the PVS sequent and the SAL conjecture afforded by the expressiveness of SAL's language. The main difference between the sequent and conjecture are the use of arrays in SAL rather than curried functions in PVS and that the number of nodes are fixed in the finite-state SAL specification.

For a fixed number of nodes, SAL easily returns a concrete counterexample showing how a state can be reached in which the theorem is false.

```
counterex : CLAIM system |-
  G(   (pc = 4 AND
        r_status[1] = good AND
        G_status = asymmetric AND
        IC_DMFA(r_status, F_RB, F_BR, G_status) AND
        all_correct_accs(r_status, F_RB,
                         G_status, F_BR, F_BB))
    =>
       (F_BR[1] = trusted OR
        F_BB[2] = declared OR
        ((FORALL (r: RMUs): F_RB[1][r] = trusted =>
              r_status[r] /= asymmetric)
         AND
         (FORALL (r: RMUs): F_RB[2][r] = trusted =>
              r_status[r] /= asymmetric)) OR
        F_BB[1] = declared OR
        robus_ic[1] = robus_ic[2]));
```

Figure 2: SAL Formulation in LTL of the Unproved Sequent

While our case-study highlights the benefit of interactivity between model checking and theorem proving, further work is required. The case-study suffers at least the following shortcomings:

- The approach is too interactive and onerous. It requires manually specifying the protocol and failed conjecture in a model checker and manually correcting the specification in the theorem prover.
- The approach depends on the counterexample being attainable with instantiated parameters that are small enough to be model checked. As pointed out by an anonymous reviewer of this report, that the counterexample was uncovered with small finite values accords with Daniel Jackson's "small scope hypothesis" [14]. For the case presented, we could have uncovered the error through model checking alone, but our goal was to prove the protocols correct for any instantiation of the parameters, as we were in fact able to do, once the protocol was mended [17].
- We would like a more automated approach to verify the parameterized protocol specification in the first place than is possible using mechanical theorem proving alone.

A more automated connection between PVS and SAL would be a good start to satisfying many of these desiderata.

6.2 Real-Time Schedule Verification

In this final example, I used PVS to specify and verify a general mathematical theory, and then I used SAL to automatically prove that various hardware realizations satisfied the theory's constraints. Specifically, I used PVS to extend a general theory about time-triggered systems. *Time-triggered systems* are distributed systems in which the nodes are independently-clocked but maintain synchrony with one another. Time-triggered protocols depend on the synchrony assumption the underlying system provides, and the protocols are often formally verified in an untimed or synchronous model based on this assumption. An untimed model is simpler than a real-time model, but it abstracts away timing assumptions that must hold for the model to be valid.

John Rushby developed a theory of time-triggered system in PVS [23]. The central theorem of that work showed that under certain constraints, a time-triggered system simulates a synchronous system. Some of the constraints (or axioms, as they were formulated in PVS) were inconsistent. I mended these inconsistencies and extended the theory to include a larger class of time-triggered protocols [20].

A theorem prover was the right tool for this effort, as the theory is axiomatic and its proofs rely on a variety of arithmetic facts (e.g., properties of floor and ceiling functions, properties of absolute values, etc.). Furthermore, using PVS, I could formally prove the theory consistent by showing a model exists for it using theory interpretations, as implemented in PVS [19].

Once the theory was developed, I wished to prove that specific hardware schedules satisfied the real-time constraints of the theory. To do so, I essentially lifted the schedule constraints (i.e., the axioms) from the PVS specification into SAL, given the similarity of the languages. Then I built a simple state machine that emulated the evolution of the hardware schedules through the execution of the protocol. I finally proved theorems stating that in the execution of the state machine, the constraints are never violated. This verification also used inf-bmc (Section 4), since the constraints were real-time constraints.

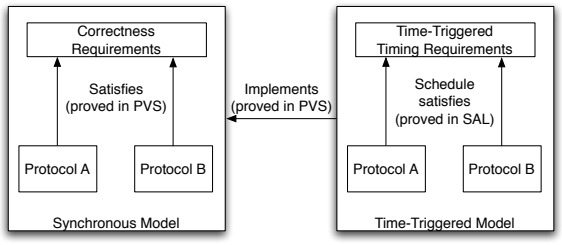


Figure 3: Time-Triggered Protocol Verification Strategy

The upshot of the approach is a formally-verified connection between the untimed specification and the hardware realization of a time-triggered protocol with respect to its timing parameters, as shown in Figure 3.

Stepping back, the above approach is an example of the judicious combination of mechanical theorem proving and model checking. While theorem proving requires more human guidance, it was appropriate for formulating and verifying the theory of time-triggered systems because the theory required substantial mathematical reasoning, and we only have to develop the theory once. To prove the timing characteristics of the implementations are correct, model checking was appropriate because the proofs can be automated, and the task must be repeated for each implementation or

optimization for a single implementation.

7. CONCLUSION

My goal in this paper was to advocate for and demonstrate the utility of the advanced features of SAL. I hope this report serves as a "cookbook" of sorts for the uses of SAL I have described.

In addition to the features and techniques I have demonstrated herein, other applications have been developed. As one example, Grégoire Hamon, Leonardo de Moura, and John Rushby prototyped a novel automated test-case generator in SAL [11]. The prototype is a few-dozen line Scheme program that calls the SAL API for its model checkers. Still other uses can be found on the SAL wiki at `http://sal-wiki.csl.sri.com/index.php/Main_Page`, which should continue to provide the community with additional SAL successes.

Acknowledgments

Many of the best ideas described herein are from my coauthors. I particularly thank Geoffrey Brown for his fruitful collaboration using SAL. I was first inspired to use SAL from attending Bruno Dutertre's seminar at the National Institute of Aerospace. John Rushby's SAL tutorial [25] helped me enormously to learn to exploit the language. I received detailed comments from the workshop's anonymous reviewers and from Levent Erkök at Galois, Inc. Much of the research cited herein was completed while I was a member of the NASA Langley Research Center Formal Methods Group.

8. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] C. Barrett, L. de Moura, and A. Stump. Design and results of the 2nd satisfiability modulo theories competition (SMT-COMP 2006). *Formal Methods in System Design*, 2007. Accepted June, 2007. To appear. A preprint is available at `http://www.smtcomp.org/`.

[3] G. Brown and L. Pike. Temporal refinement using smt and model checking with an application to physical-layer protocols. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE'2007)*, pages 171–180. OmniPress, 2007. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/refinement.html`.

[4] G. M. Brown and L. Pike. Easy parameterized verification of biphase mark and 8N1 protocols. In *The Proceedings of the 12th International Conference on Tools and the Construction of Algorithms (TACAS'06)*, pages 58–72, 2006. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/bmp.html`.

[5] G. M. Brown and L. Pike. "easy" parameterized verificaton of cross domain clock protocols. In *Seventh International Workshop on Designing Correct Circuits DCC: Participants' Proceedings*, 2006. Satellite Event of ETAPS. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/dcc.html`.

[6] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.

[8] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In *Computer-Aided Verification, CAV'03*, volume 2725 of *LNCS*, 2003.

[9] B. Dutertre and L. de Moura. Yices: an SMT solver. Available at `http://yices.csl.sri.com/`, August 2006.

[10] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214, Grenoble, France, Sept. 2004. Springer-Verlag. Available at `http://fm.csl.sri.com/doc/abstracts/ftrtft04`.

[11] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270, Beijing, China, Sept. 2004. IEEE Computer Society.

[12] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[13] D. V. Hung. Modelling and verification of biphase mark protocols using PVS. In *Proceedings of the International Conference on Applications of Concurrency to System Design (CSD'98), Aizu-wakamatsu, Fukushima, Japan, March 1998*, pages 88–98. IEEE Computer Society Press, 1998.

[14] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[15] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, July 1982. Available at `http://citeseer.nj.nec.com/lamport82byzantine.html`.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] P. Miner, A. Geser, L. Pike, and J. Maddalon. A unified fault-tolerance protocol. In Y. Lakhnech and S. Yovine, editors, *Formal Techniques, Modeling and Analysis of Timed and Fault-Tolerant Systems (FORMATS-FTRTFT)*, volume 3253 of *LNCS*, pages 167–182. Springer, 2004. Available at `http://www.cs.indiana.edu/~lepike/pub_pages/unified.html`.

[18] J. S. Moore. A formal model of asynchronous communication and its use in mechanically verifying a biphase mark protocol. *Formal Aspects of Computing*, 6(1):60–91, 1994.

[19] S. Owre and N. Shankar. Theory interpretations in PVS. Technical Report SRI-CSL-01-01, SRI, International, April 2001. Available at `http://pvs.csl.sri.com/documentation.shtml`.

[20] L. Pike. Modeling time-triggered protocols and verifying their real-time schedules. In *Proceedings of Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007. Available at `http://www.`

cs.indiana.edu/~lepike/pub_pages/fmcad.html. To appear.

[21] L. Pike and S. D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 286–289, New York, NY, USA, 2005. ACM Press. Available at http://www.cs.indiana.edu/~lepike/pub_pages/emsoft.html.

[22] L. Pike, P. Miner, and W. Torres. Model checking failed conjectures in theorem proving: a case study. Technical Report NASA/TM–2004–213278, NASA Langley Research Center, November 2004. Available at http://www.cs.indiana.edu/~lepike/pub_pages/unproven.html.

[23] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

[24] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag. Available at http://www.csl.sri.com/users/rushby/abstracts/cav00.

[25] J. Rushby. SAL tutorial: Analyzing the fault-tolerant algorithm OM(1). Technical Report CSL Technical Note, SRI International, 2004. Available at http://www.csl.sri.com/users/rushby/abstracts/om1.

[26] J. Rushby. Harnessing disruptive innovation in formal verification. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. IEEE Computer Society, 2006. Available at http://www.csl.sri.com/users/rushby/abstracts/sefm06.

[27] W. Torres-Pomales, M. R. Malekpour, and P. Miner. ROBUS-2: A fault-tolerant broadcast communication system. Technical Report NASA/TM-2005-213540, NASA Langley Research Center, 2005.

[28] F. W. Vaandrager and A. L. de Groot. Analysis of a Biphase Mark Protocol with Uppaal and PVS. Technical Report NIII-R0455, Nijmegen Institute for Computing and Information Science, 2004.

Modelling and test generation using SAL for interoperability testing in Consumer Electronics

Srikanth Mujjiga
Philips Research Asia - Bangalore
srikanth.mujjiga@philips.com

Srihari Sukumaran
Philips Research Asia - Bangalore
srihari.sukumaran@philips.com

ABSTRACT

Testing of consumer electronics (CE) devices for interoperability with respect to standards is an important validation activity. We have developed a model-based approach for producing interoperability tests based on a standards specification. This involves manually constructing individual device models in SAL, based on the standards, and using the `sal-atg` tool to generate tests from the composed system model. We describe this approach using an example standard. We also point out a problem with this approach, that has to do with variability in the devices due to optional and vendor-specific features. We propose an extension to the SAL language that addresses this problem. The extension is designed to enable the continued use of SAL tools for analysis purposes.

1. INTRODUCTION

Interoperability (refer [12],[19] etc.) is the ability of entities to communicate and use each others published capabilities to offer the required functionalities to the end users. In the context of consumer electronic (CE) devices, interoperability is typically in the context of some standard. We will consider a set of CE devices to be interoperable with respect to some standard if they can interact to provide a user the set of functionalities defined by the standard. Interoperability testing is a key validation activity for entities that are to function as part of a large system. Interoperability testing tests the end-to-end functionality in a complete system of multiple devices to validate that they are interoperable.

In prior work [17, 13] we have described the model-based approach we have taken to generate interoperability tests. We used the SAL language and `sal-atg` tool to realise our approach. An important reason for choosing SAL was the availability of a powerful, flexible and configurable model-based test generation tool like `sal-atg`[7, 8] that allows generation of traces to cover a user given set of goals. This is very useful in realising our approach [13]. SAL language is also quite appropriate for modelling state-machine based device behaviours as specified by the CE standards. The presence of a large number of data types (arrays, records, etc.) and the ability to specify state machines in a compositional manner and using a guard-action style are quite appropriate for our needs. In this paper we will first present our past and ongoing work on modelling and generating tests based on the CE standards using SAL. We will use an example (the CEC standard) to illustrate what we had to do in terms of modelling in SAL. We will then describe some limitations and potential problems that arise in the context of interoperability testing due to the presence of great variability due to optionality in standards and vendor-specific extensions. We propose a slight extension to the SAL language that addresses our problems and give an outline of how we can use this extended SAL language to generate tests (using standard `sal-atg` itself).

The paper is organized as follows: In section 2 we give some background on interoperability and the CEC specification. In section 3 we describe the modelling of CEC devices and automated interoperability test case generation and the problem we faced in the modelling step. Section 4 gives the extensions we propose in the SAL modelling language. In section 5 we discuss how to fit modelling in the extended SAL language into the test generation process using `sal-atg`. Conclusion and future work is presented in section 6.

2. BACKGROUND

2.1 Interoperability test generation for CE

In interoperability testing, tests are run to check whether a system of two or more devices interoperate with each other. The test cases for interoperability testing depends on the system set-up which is normally given by the test engineer. Often there are many set-ups that are to be tested and manual test construction for each setup is tedious and error prone and also there is no complete test coverage guarantees. To address this problem we have developed an automatic interoperability test generation framework based on model based testing [13]. We construct models for each device type from the standards specification. This process is manual and requires some skill, since the standards specifications are not completely formal; they are typically in structured English text. We also chose to write the models directly in the SAL language, since as described earlier the language mostly provides all that we need. The SAL device models are composed based on the test set-up given by the test engineer and finally `sal-atg` is used on the composed model for automatic test generation.

AFM '07, November 6, Atlanta, GA, USA.

In section 3 we will describe the modelling aspects of the above approach, for the CEC standard, in some more detail.

2.2 Consumer Electronics Control (CEC)

The High-Definition Multimedia Interface (HDMI) [1] is used to transmit digital audio-visual signals from source devices like DVD players, set-top boxes, etc., to sink devices like television sets, projectors and other displays. HDMI is capable of carrying high quality multi-channel audio data and all standard and high-definition consumer electronics video formats. HDMI includes three separate communication channels, TMDS channel for carrying all audio and video data as well as auxiliary data, the DDC channel, which is used by an HDMI source to determine the capabilities and characteristics of the sink and the CEC channel which is used for high-level user functions such as automatic setup tasks or tasks typically associated with infrared remote control usage.

CEC is a protocol that provides features to enhance the functionality and interoperability of devices within an HDMI system. CEC is a protocol based on a bus system. All devices which conform to the CEC standard (called CEC devices henceforth) have both a physical and a logical address. The logical address defines a device type as well as being a unique identifier. There are 16 possible logical addresses in a CEC network. CEC specification defines five device types and they are TV, playback device, recording device, tuner device and audio system. A CEC device is called a source device if it is currently providing an audio-video stream via HDMI (example: DVD Player) or a sink device if it has an HDMI input and the ability to display the input HDMI signal (example: TV).

CEC (version 1.3a) provides nine end-user features and each feature is provided by a set of messages and associated behaviours. Some CEC messages require parameters and the CEC message along with its parameters is embedded in a CEC frame. There are 62 CEC messages and CEC specification specifies how a device should respond to the message if it wants to support it. CEC devices send messages as response to some message it has received or a remote control operation by the user. There are two types of CEC messages – broadcast messages to which all the CEC devices must respond appropriately and directed messages which are specifically directed toward a single CEC device. There can be at most one CEC frame (and hence one CEC message) at any time on the CEC channel (bus) and this is ensured by CEC arbitration. A CEC device in case of an unsupported message directed to it will respond by sending a special message `Feature Abort` on the CEC channel. Some of the CEC messages require pre-defined parameter values and CEC device's response to some of these parameter values are optional. For example, the CEC message `play` requires one parameter and the CEC specification define 15 possible values that this parameter can take. But only the response for parameter `Play Forward` is mandatory to implement message `play` and all other 14 parameters are optional. CEC devices will respond with `Feature Abort` if they do not support them. So it is again up to the vendor to decide whether to implement it or not.

3. MODELING AND TEST GENERATION FOR CEC DEVICES

3.1 Construction of models for CEC device types

CEC specification defines five device types based on functionality and we model each of these device types as SAL base modules. All the nine end-user features involving the 62 CEC messages are modelled. So these device type base modules include all the mandatory and optional features given in the CEC specifications. To illustrate, a part of the SAL module of the playback device, including only the `One Touch Play` and `Device SOD Name Transfer` features, is shown in figure 1. The `One Touch Play` feature allows a device to be played and become the active source with a single button press. `Active source`, `Image View On` and `Text View On` are the CEC messages that are used in this feature. CEC devices use `Device OSD Name Transfer` feature, which involves two CEC messages `Give OSD Name` and `Set OSD Name`, to request or send the preferred name to be used in any on-screen display (e.g., menus). See [1] for more details.

The CEC frame is modeled as a SAL record as shown in the lines 5 to 8 in figure 1. The parameters to the CEC messages are abstracted to integers in the range 0 to 15 as shown in line 7. The `source` field in the CEC frame is the logical address of the device which is initiating the CEC message and `destination` field is the logical address of the device to which the message is directed. A special address `15` in the destination field indicates that it a broadcast message. Every CEC device will have a unique logical address based on its functionality (device type). So all the device type based modules are parameterized by their logical address as shown in line 11.

CEC devices respond to CEC message on the CEC channel and the possible responses are given in the CEC specifications. The response to a CEC message may require a CEC device to initiate other CEC messages on the CEC channel. Devices may also initiate CEC messages as a result of some remote control operations from the user. The SAL base module for each CEC device type gets the CEC frame on the CEC channel via its `INPUT` variable `I_msg_blk` and initiates a CEC messages via its `OUTPUT` variable `O_msg_blk`. A special value `Null` in the `msg` field of `O_msg_blk` indicates that the device is not initiating any CEC message. Similarly, the value `Null` in the `msg` field of `I_msg_blk` implies that there is no CEC frame currently on the CEC channel, i.e., the CEC channel is currently idle.

A CEC device is either in `On` or `Standby` state (which is defined as a SAL `TYPE` in line 9). A playback device can also be in either `Deck Active` or `Deck Inactive` state (defined at line 10). CEC specifications gives the possible states of all the device types. Initially all the devices are assumed to be in `On` state and playback devices are assumed to be in `Deck Inactive` state (as shown in line 17 of figure 1. We have used only SAL SimpleDefinitions[11] for variable initializations. There is a NamedCommand[11] transition per valid value of each CEC messages. For example, lines 21 to 24 in figure 1 shows the response of the playback device for the incoming CEC message `Active Source`. If the device response to a CEC messages does not result in it initiating any CEC message, then the `msg` field in its `OUTPUT` variable is assigned a special value `Null` as shown in line 24. CEC Devices can also initiate a CEC messages in response to some remote control operation given by the user. For example, a remote control `play` on a CEC playback device like a DVD player may result in the DVD player becoming the active

source (if it is in the position to become active source). In such situation DVD player will respond by sending either `Image_View_On` or `Text_View_On` message directed to the TV (the logical address of TV is `0`). We have modelled the initiation of messages on the CEC channel caused by the external environment (like remote control, user pressing a button on the device, etc.) as non-deterministic transitions as shown in line 26 to 46 in figure 1. Lines 21 to 46 in figure 1 are the transitions for messages required for `One Touch Play` feature and the lines 48 to 63 are for the `Device OSD Name Transfer` feature.

3.2 Construction of Bus model

CEC is based on a bus system and all the transactions on the CEC bus consist of an initiator and one or more followers. The initiator sends the CEC frame (CEC message and data) and the followers respond to the CEC message. There can be only one CEC frame at any time on the CEC channel. The CEC bus arbitration ensures collisions are spotted and reliable messaging can be achieved. It does this in a completely static priority-based manner based on the device's logical address. The bus system of CEC is modeled as a SAL base module. Currently we have modeled a simple bus. The SAL code for a bus module which connects one sink and two sources is shown in figure 2. This can easily be generalised to the maximum number of sources.

The bus module has one `OUTPUT` variable `out_block` and an array of `INPUT` variables `in_block` as shown in lines 3 and 4 of figure 2. It accepts the `OUTPUT` variable of all CEC devices (`O_msg_blk`) via its `INPUT` variables. Depending on device priorities, the bus module copies the value of one of its `INPUT` variable into its `OUTPUT` variable `out_block`. As per the standard, the bus module gives highest priority to the device with lowest value as its logical address. The `OUTPUT` variable `out_block` of the bus module is made as `INPUT` to all the CEC device modules (this is done by renaming `OUTPUT` variable of bus module to `I_msg_blk`). As shown in the guards of the transitions in the bus module in figure 2, the value of the `in_block` corresponding to higher priority device is copied into `out_block` only if the CEC device initiated a valid messages, i.e., only if `msg` field is not `Null`. Similarly for the lower priority devices the value of `in_block` is copied into `out_block` only if the above condition is true and none of the higher priority devices have any messages, i.e., their `in_blocks` are `Null`.

3.3 Construction of SUT model

The test set-up for interoperability testing will be a set of CEC devices under test connected together by the CEC bus. So the module for the system under test (SUT) is constructed by instantiating the device type base modules corresponding to the actual device under test (given by the test set-up information) and composing them synchronously with the bus module. The pictorial view of the composition of a sample test set-up with 1 TV and 2 play back devices (DVD players) is shown in figure 3 and the corresponding SAL module is shown in figure 4. Renaming is done to make the `OUTPUT` of the bus module as `INPUT` to the device modules as shown in line 14 of figure 4 and `OUTPUT` of the device modules as `INPUT` to the bus module as shown in lines 3, 6, 9 and 13.

3.4 Test generation

```
1  CEC : CONTEXT = BEGIN
2  CEC_Msg : TYPE = { Null, Active_Source,Image_View_On,
3  Text_View_On,Set_OSD_Name,Give_OSD_Name,feature_abort};
4  Dev_Addr : TYPE = [ 0 .. 15 ];
5  Msg_Blk : TYPE = [#
6  msg : CEC_Msg,              %%% Actual message
7  param1 : [0..15],           %%% Abstracted parameter
8  source : Dev_Addr, destination : Dev_Addr #];
9  State1 : TYPE = { On, Standby };
10 PBD_State2 : TYPE = { Deck_Active, Deck_Inactive };
11 PBD[my_la : types!Dev_Addr] : MODULE = BEGIN
12 INPUT I_msg_blk : types!Msg_Blk;
13 OUTPUT O_msg_blk : types!Msg_Blk
14 LOCAL cs1 : State1, cs2 : PBD_State2, as : BOOLEAN,
15 curr_active_source : Dev_Addr,
16 INITIALIZATION
17 cs1 = On; cs2 = Deck_Inactive; O_msg_blk.msg = Null;
18 as = FALSE; curr_active_source = 0;
19 TRANSITION
20 [%%% Feature 1 : One Touch Play
21 ACTIVE_SOURCE_I: I_msg_blk.msg = Active_Source
22 AND cs1 = On  -->
23 curr_active_source' = I_msg_blk.param1;
24 O_msg_blk'.msg = Null;
25 []
26 IMAGE_VIEW_ON_O: I_msg_blk.msg = Null AND
27 cs1 = On-->
28 O_msg_blk'.msg = Image_View_On;
29 O_msg_blk'.source = my_la;
30 O_msg_blk'.destination = 0;
31 as' = TRUE;
32 []
33 TEXT_VIEW_ON_O: I_msg_blk.msg = Null AND
34 cs1 = On -->
35 O_msg_blk'.msg = Text_View_On;
36 O_msg_blk'.source = my_la;
37 O_msg_blk'.destination = 0;
38 as' = TRUE;
39 []
40 ACTIVE_SOURCE_O: I_msg_blk.msg = Null AND
41 cs1 = On AND as = TRUE -->
42 O_msg_blk'.msg = Active_Source;
43 O_msg_blk'.source = my_la;
44 O_msg_blk'.param1 = my_la;
45 O_msg_blk'.destination = 15;
46 as' = FALSE;
47 []
48 %%% Fature 2 : Device OSD Name Transfer
49 GIVE_OSD_NAME_I: I_msg_blk.msg = Give_OSD_Name
50 AND cs1 = On -->
51 O_msg_blk'.msg = Set_OSD_Name;
52 O_msg_blk'.source = my_la;
53 O_msg_blk'.destination = I_msg_blk.source;
54 O_msg_blk'.param1 = my_la;
55 []
56 SET_OSD_NAME_I: I_msg_blk.msg = Set_OSD_Name
57 AND cs1 = On -->
58 O_msg_blk'.msg = Null;
59 []
60 GIVE_OSD_NAME_O: I_msg_blk.msg = Null
61 AND cs1 = On -->
62 O_msg_blk'.msg = Give_OSD_Name;
63 O_msg_blk'.source = my_la;
64 []
65 ELSE --> O_msg_blk'.msg = Null;
66 ]END;
```

Figure 1: SAL code for CEC playback device

Boolean trap variables are inserted in appropriate locations in the base modules corresponding to CEC devices in

```
BUS : MODULE =
BEGIN
INPUT in_block : ARRAY [0..2] OF Msg_Blk
OUTPUT out_block : Msg_blk
INITIALIZATION
 out_block.msg = Null;
TRANSITION
[
 in_block[0].msg /= Null -->
  out_block' = in_block[0];
 []
 in_block[0].msg = Null AND in_block[1].msg /= Null -->
  out_block' = in_block[1];
 []
 in_block[0].msg = Null AND in_block[1].msg = Null
  AND in_block[0].msg /= Null -->
  out_block' = in_block[2];
 []
 ELSE -->
]
END;
```

Figure 2: SAL code for bus module

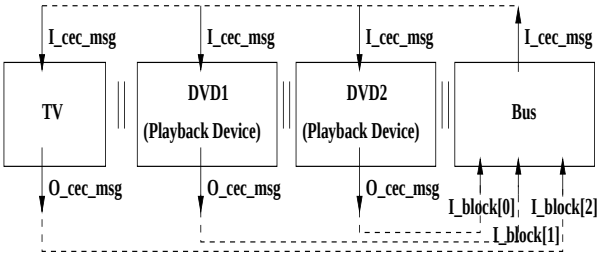


Figure 3: Model composition for a sample test set-up

```
CEC_SUT : MODULE =
(WITH OUTPUT I_block : ARRAY [0..2] OF Msg_Blk
 RENAME O_msg_blk to I_block[0]
 IN TV[0]  %%% TV with logical address 0
 ||
 RENAME O_msg_blk to I_block[1]
 IN PBD[4]  %%% DVD1 with logical address 4
 ||
 RENAME O_msg_blk to I_block[2]
 IN PBD[5])  %%% DVD1 with logical address 8
 ||
(WITH OUTPUT I_msg_blk : Msg_Blk;
      RENAME out_block to I_msg_blk,
   in_block to O_msg_blk IN BUS);
END
```

Figure 4: SAL code for a sample test set-up

the test set-up for test generation based on the test coverage criteria. One trap variable for each transition is enough for whole model transition coverage. Test cases can then be generated automatically by using `sal-atg` on the SUT module. The presence of a flexible, "off-the-shelf" test generator like `sal-atg` was one of the great advantages of using the SAL language and tools. For our models we have no problems with getting full coverage though sometimes some tuning of the parameters `-id`, `-ed` and `-md` (especially the last two) is required to get good tests that give full coverage with as little redundancy as possible. When `ICS` was the solver, in the case of the models for some standards (which are larger and more complicated than CEC) there was some issues with ICS running out of memory. After upgrading to the `yices` solver this has not been an issue at all and the generation times are also quite insignificant.

Device	Severe bugs	Moderate bugs
DVD player 1	3	3
DVD player 2	0	2
TV 1	0	3

Table 1: Summary of bugs found

4. DISCUSSION ON CEC TEST GENERATION

In this section we will first summarise the results of applying the approach described in the previous section to a few devices. We will then discuss in detail some of the practical issues in directly applying the approach in engineering practice. This will set the stage for the rest of this paper.

4.1 Results

We ran the tests generated as described above on a number of devices. So far for CEC we have only tested one device at a time. The other device (sink in case the device under test – DUT – is a source and vice-versa) is simulated via an internally developed CEC device simulation tool. The DUT and the simulator communicate on a CEC bus. The simulator can also simulate UI commands (remote control) to the DUT. Currently the tests are manually translated to commands for the simulator (to simulate sink/source and UI commands); but we can automate this since the simulator supports a C like scripting language.

We have tested DUTs (specifically we test the CEC implementations of these devices) that are sources (DVD players) and sinks (TV); the devices were of different makes, some already on the market. In all cases we have detected behaviours that are deviations from the CEC specification. Some of these are severe – in terms of potential impact of the deviation, as per the standards classification used in Philips CE (note that, since we are dealing with CE products, the impact is typically in terms of the cost to fix or market impact, rather any safety or life criticality). Table 1 summarises the bugs that were found. It is worth mentioning again that all the devices we tested had gone successfully through the complete product testing cycle, and hence what we have caught are residual bugs.

4.2 Issues and possible solutions

The above approach becomes rather unwieldy in practise since the device that are under test typically do not implement all the feature or support all the messages given in the standards. Optional features are very common in CE standards and the optional features supported by the actual devices under test vary depending on the vendors and the model. Similarly many standards (though not CEC itself) allow vendor specific extensions. It is very important to generate tests based on models that accurately capture the device functionality.

The problem here is that the supported optional features and vendor extensions are specific to each actual device and

so it is not possible to have one generic model for each device type which can just be instantiated for all the corresponding actual devices. Let us examine this using an example: the sample test set-up shown in figure 3. Consider the two play back devices, DVD1 and DVD2 in the test set-up. DVD1 supports both `One Touch Play` and `Device OSD Name Transfer` features but it implements only `Image View On`, i.e., DVD1 can only send `Image View On` when it wants to become a source. It does not violate CEC specification because it is in accordance to specification – "it is mandatory for a source to implement at least one of `Image View On` or `Text View On`". DVD2 implements only the `One Touch Play` feature (implements all the 3 messages) and does not support `Device OSD Name Transfer`. In such a situation we cannot use the base module of the play back device we constructed from the standards since it includes all the features. It requires modelling of each device under test as a separate base module; however this is inefficient since all the devices have lots of features in common. Also in such case, changes in CEC specifications will require changes in all the device modules.

One possible solution is to parametrize the device type base modules with boolean parameters; one for each optional feature (and message). If the value of the parameter is `TRUE` then it implies that the device implements the corresponding optional message, else it feature aborts. Every guard will then have to check the value of the boolean parameter corresponding to some optional feature and/or message. The guard which checks for the value `TRUE` will have the transition with assignments which will implements the optional message and the the guard which checks for the value `FALSE` will have assignments which correspond to the device `feature abort`ing the CEC message. So there will be two transitions for each optional message, one which implements it and other which feature aborts. The SUT module is then constructed by instantiating the based modules by passing appropriate parameter values based on the optional features supported by the actual devices under test and synchronously composing these instantiated based modules with the bus module. But CE standards with many optional features will require a huge parameter list and the device module instances will not be very intuitive. A more severe problem is to do with vendor specific extension features and in such situations this solution will not work as it is not possible to include new vendor specific feature in the modules by instantiating parameter values since the device type base module does not know about these vendor specific features. It will require new transitions and may also require new state variables and so we will have to write base modules for each device under test that includes vendor specific extension features.

Basically there are four types of features in any device model:

1. those retained from the device type model (i.e., from the standard),
2. those (optional) device type features that are not implemented,
3. vendor specific extensions of device type features, and
4. vendor specific new features.

Using module paramters one can handle 1. and 2., but 3. and 4. cannot be handled this way.

A better and an efficient solution is to "derive" from the device type models (constructed from standards) the models of the actual devices under test by "augmenting" with optional and vendor specific features they support. One way to implement this is to have constructs in the modelling language that will allow deriving new models from existing models by declaring new variable and new transitions. We plan to implement such a solution by extending the SAL language with new constructs (called extended SAL henceforth) that will allow new modules to be derived from existing modules, and having a translator from this extended SAL to "basic" SAL. The proposed extensions and the translation procedure is explain in the next section.

The new constructs introduced are to ease the task of model construction, specifically for devices based on standards in CE domain. At this point of time we cannot comment any thing about the advantages of these SAL extensions for other modelling related tasks. For standards in CE domain, these extensions will facilitate easy model construction and maintenance of models. If there are any changes in standards, then only the base modules need to be changed.

5. EXTENDED SAL

We extend the SAL language by allowing modules to be defined as extensions of already existing base modules using the keyword `EXTENDS`. Such an extended module will be called extended base module. Note that we do not increase the expressability of the SAL language. This means that every extended base module can be translated to an equivalent base module and hence the standard SAL tools are all applicable. Table 2 gives those part of the grammar for extended SAL that is different from that of SAL (as given in [11]).

Basically we are introducing a new type of module: the ExtendedBaseModule. It is defined as extending (keyword `EXTENDS`) a base module and defining additional variables, initialisations, and transitions. The initialisations introduced in the extended base module can only be SimpleDefinitions and similarly the transitions can only be NamedCommands. It should be noted that these restrictions are imposed because of the modelling style we use for modelling in the CE space. It is possible that our concept of extended base modules will work in the general setting also, but it provides no value for us for constructing and managing models in the CE world.

Every extended base module is equivalent to a base module. We describe this equivalence by explaining how to translate a extended base module to its equivalent base module. Let a module `M1` extend a base module `M2`. Then the extended base module `M1` can be translated to a base module `M1'` in the following manner:

- The module parameters of `M1'` is the union of the module parameters of `M2` and the parameters given in the declaration of `M1`. In case `M1` declares a module parameter that is already present in `M2` then an error is flagged.
- Variables are handled exactly as above, i.e., the variables of `M1'` is the union of the variables of `M2` and the variables given in the declaration of `M1`. In case `M1` declares a variable that is already present in `M2` then an error is flagged.

Module	:=	BaseModule
		| ModuleInstance
		| SynchronousComposition
		| AsynchronousComposition
		| MultiSynchronous
		| MultiAsynchronous
		| Hiding
		| NewOutput
		| Renaming
		| WithModule
		| ObserveModule
		| (Module)
		| ExtendedBaseModule
ExtendedBaseModule	:=	`EXTENDS` Identifier `BEGIN` ExtendedBaseDeclarations `END`
ExtendedBaseDeclarations	:=	{ ExtendedBaseDeclaration }*
ExtendedBaseDeclaration	:=	InputDecl
		| OutputDecl
		| GlobalDecl
		| LocalDecl
		| ExtendedInitDecl
		| ExtendedTransDecl
ExtendedInitDecl	:=	`INITIALIZATION` { SimpleDefinition }+; [;]
ExtendedTransDecl	:=	`TRANSITION` { [ExtendedSomeCommands] }+; [;]
ExtendedSomeCommands	:=	{ NamedCommand }+ [] [[]ElseCommand]

Table 2: Grammar

- The initialisations of `M1'` consists of the initialisations of `M1` and those initialisations of `M2` that are not overridden in `M1`.

- The transitions of `M1'` consists of the transitions of `M1` and those transitions `g -> a` of `M2` such that `M1` does not have a transition `g' -> a'` where `g` is equivalent to `g'`.

An example in figure 5 shows a base module (`calc`, for a simple calculator) and an extended base module that extends from it (`calc_Vendor_A`). Figure 6 shows the base module (`calc_Vendor_A_trans`) that is equivalent to the extended base module in 5. This was obtained by hand-translating the module `calc_Vendor_A` as per the strategy given above. We are in the process of formalising the strategy and implementing it in an automatic translator.

In the example the base module `calc` has three features: `ADD`, `DIVD`, and `MOD` (optional). The extended base module `calc_Vendor_A` introduces a new (vendor-specific) feature `SUB` and extends the definition of feature `DIVD`. It also does not implement the optional feature `MOD`. The module `calc_Vendor_A_trans` shows how these have been translated to a SAL base module. This indicates how all the four types of features in a device model (as enumerated in the preceding section) can be specified using the extended SAL language. Basically the ability to define new transitions and extend existing ones by redefining the action enables us to model the devices in a clean manner.

Note that in the example we only define initialisations and transitions in the extended module `calc_Vendor_A`. This is infact representative of the way we would actually use the extended SAL language, since the most common use does not include introducing new variables.

```
1  calc : MODULE  = BEGIN
2   INPUT A : INTEGER, B : INTEGER
3   INPUT Ope : operations
4   OUTPUT Out : INTEGER
5  INITIALIZATION
6   Out = -1;
7  TRANSITION
8  [
9   Ope = ADD --> Out' = A + B;
10  []
11  Ope = DIVD --> Out' = A div B;
12  []
13  Ope = MOD --> Out' = A mod B; % optional
14 ]END;
15
16 calc_Vendor_A : MODULE = EXTENDS calc BEGIN
17 INITIALIZATION
18  Out = 0;
19 TRANSITION
20 [
21  Ope = SUB --> Out' = A - B;   % vendor new feature
22  []
23  Ope = DIVD --> Out' =        % vendor extension
24    IF B = 0 THEN -1 ELSE A div B ENDIF;
25  []
26  Ope = MOD --> ;               % unsupported optional
27 ]END;
```

Figure 5: A sample base module and extended base module in extended SAL

6. USING EXTENDED SAL FOR MODELLING AND GENERATING TESTS

We now discuss how we propose to use the extended SAL language in the interoperability test generation process. Figure 7 gives the entire process. As shown in the figure, first, base modules for each device type are constructed from the standards. These device type base modules cover all the

```
calc_Vendor_A_trans : MODULE = BEGIN
 INPUT A : INTEGER, B : INTEGER
 INPUT Ope : operations
 OUTPUT Out : INTEGER
INITIALIZATION
 Out = 0;
TRANSITION
[
 Ope = ADD --> Out' = A + B; % retained from base
 []
 Ope = SUB --> Out' = A - B; % new
 []
 Ope = DIVD --> Out' =       % extension
   IF B = 0 THEN -1 ELSE A div B ENDIF;
 []
 Ope = MOD --> ;             % not supported
]END;
```

Figure 6: The translated SAL base module

standard-specified features, both mandatory and optional. This is manual (indicated by the blue colour in the figure) as it depends on the standards under interest. The second step is to insert boolean trap variables in appropriate locations in the device type base modules depending on the test coverage criteria. This step can be fully automated. The dashed arrow indicates an transformation or a generation step, thus the device type modules with goals can be generated from the device type models and the test criteria. The above parts of the process is exactly as earlier (section 3).

The next step is to construct an extended base module model for each device under test by extending the corresponding device type base module (with trap variables) based on the supported features (implemented/unimplemented optional features, vendor specific features if allowed by standard, etc.). This will generally have to be a manual process – it can be automated if the device models only implement or not-implement the optional features and do not add any new vendor specific features. The blue colour again indicates it is a manual process; also the solid arrow indicate a usage of one artifact (the base of the arrow) in the other (the head of the arrow). The module for SUT is obtained by composing the device models (extended based modules) based on the test set-up information. This step can either be automated or is sometimes only a one-time activity (for some standards, e.g., UPnP [13]). Then the test generator is invoked on the SUT model to produce the test cases. This involves two steps. First (see the dashed box on the right), the translator from extended SAL is invoked as a pre-processing step. This will generate SAL base modules for all the extended base modules. Corresponding to this a new SUT module (using the generated base module names in lieu of the original extended base module names) is also produced. After this `sal-atg` is invoked, on the new SUT module produced by the translator, to generate the test cases.

Applying this approach to CEC, all the five CEC device types can be modelled as base modules based on the CEC specifications covering all the mandatory and optional features and the rest of the process as described above can be applied. The extended base modules for DVD1 and DVD2 of the example of figure 3 are shown in figure 8.

7. RELATED WORK

Test generation from behavioural models such FSMs (see

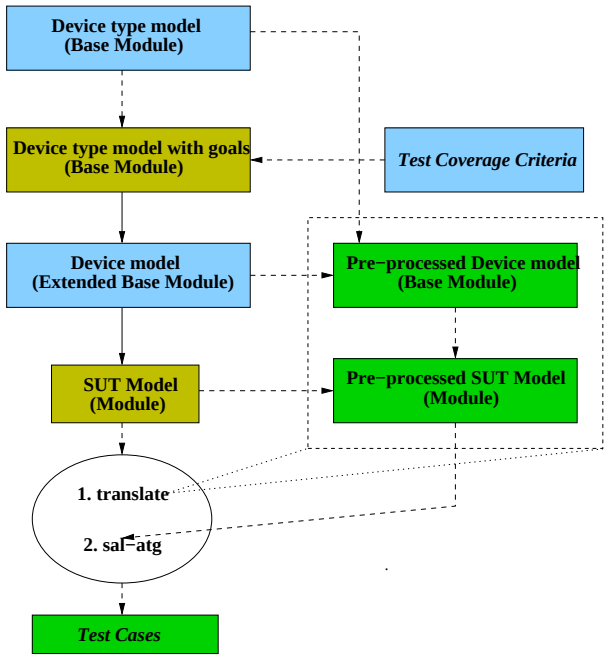


Figure 7: Approach for interoperability test case generation

```
DVD1 : MODULE = EXTENDS PBD
BEGIN
INITIALIZATION
TRANSITION
[
TEXT_VIEW_ON_O: I_msg_blk.msg = Null AND
cs1 = On --> ;
]END;

DVD2 : MODULE = EXTENDS PBD
BEGIN
INITIALIZATION
TRANSITION
[
GIVE_OSD_NAME_I: I_msg_blk.msg = Give_OSD_Name
AND cs1 = On -->
O_msg_blk'.msg = feature_abort;
O_msg_blk'.source = my_la;
O_msg_blk'.destination = 15;
[]
SET_OSD_NAME_I: I_msg_blk.msg = Set_OSD_Name
AND cs1 = On -->
O_msg_blk'.msg = feature_abort;
O_msg_blk'.source = my_la;
O_msg_blk'.destination = 15;
[]
GIVE_OSD_NAME_O: I_msg_blk.msg = Null
AND cs1 = On --> ;
]
END;
```

Figure 8: Extended base module for DVD1 and DVD2

[10]) and LTSs (see [3]) has been a very active area of research. There has been lot of work on theory, algorithms and tools. A significant fraction of this work is on testing proto-

col implementations based on FSM specifications (see [9]). There has also been lot of work (e.g., [6, 16, 18, 4]) on generating interoperability tests for network protocols. Most of these propose algorithms for test generation from state machine models based on computing reachability graphs.

Our work is similar in spirit to many of these prior work. Our focus has been to build a practical model-based test generation solution for the CE domain. We leverage SAL tools for this purpose. This enabled us to quickly come up with solutions; something that would not have been possible if we had implemented our own test generator. The ability to specify test goals provides the desired flexibility in implementing different test criteria. It is also likely that any comparable implementation that we would make will not be as efficient as `sal-atg`. Also, another important point is that in most of the above papers the models are typically just input-output state machines. In general our models are more complex than these, involving variables, and guards and actions on these variables – conceptually these are equivalent to flat state machines, but they pose significant difficulties to analyse efficiently in a symbolic manner. SAL permits us to model these naturally, and has tools to explore these models efficiently in a symbolic manner. To summarise SAL provides us both a modelling notation, and an efficient out-of-box test generator that works on user-given test goals.

There has been other work on modelling notations – specifically, Bogor [14] and Zing [2] – that offer features like the one we propose here. In fact, Bogor and Zing are meant to be translation target for model-checkers for OO software; hence they have support for inheritance and dynamic dispatch of calls. Our proposed modelling extension is more modest, but it meets our need. Moreover, Bogor and Zing do not have the test generation support offered by `sal-atg`. It is also easier to extend SAL than implement `sal-atg`-like test generation support for Bogor or Zing.

8. CONCLUSIONS AND FUTURE WORK

We have presented how we have applied model-based test generation for CE standards based interoperability testing using the SAL language and the `sal-atg` tool. We found the SAL language to be quite adequate in terms of data-structures and level of abstraction for expressing the models that we encounter. More importantly `sal-atg` is well suited for purpose. It provides automation and at the same time flexibility in the form of allowing the use of a user chosen set of test directives (from a complete set of test directives). We have had encouraging results from applying this methodology (see also [13]). We are continuing this activity and applying it to different standards in the CE space.

We also presented a particular problem with applying our approach, specifically that of modelling different device variants corresponding to a single standard device type. We propose an extension to SAL (without changing the expressability of the language) to enable easy modelling of such devices. We also described how to translate from this extended SAL to SAL. Currently we are evaluating our approach by hand translating extended SAL models to SAL models. We demonstrated our approach using one of the device types and a sub-set of the features of the CEC standard. The approach looks promising in easing the model construction and model maintenance. We plan to automate translation to enable the introduction of it into practise.

The extension we proposed is only for a subset of SAL – the subset that is of interest to us. We could consider extending it to the full SAL language. Similarly, declarations and module parameters of extended based modules cannot currently include redeclarations. This restriction can be removed, in which case the redeclared type will have to be a super-set of the base type.

9. REFERENCES

[1] High-Definition Multimedia Interface Version 1.3, 2006.

[2] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *16th International Conference on Computer Aided Verification (CAV 2004)*, pages 484–487, 2004. See `https://research.microsoft.com/projects/zing/`.

[3] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems, Advanced Lectures*, LNCS 3472. Springer, 2005.

[4] A. Desmoulin and C. Viho. Formalizing interoperability testing: Quiescence management and test generation. In F. Wang, editor, *FORTE*, volume 3731 of *Lecture Notes in Computer Science*. Springer, 2005.

[5] S. Dibuz and P. Kremer. Framework and Model for Automated Interoperability Test and its Application to ROHC. In *TestCom-03*, pages 243–257, 2003.

[6] K. El-Fakih, V. Trenkaev, N. Spitsyna, and N. Yevtushenko. FSM Based Interoperability Testing Methods for Multiple Stimuli Model. In *TestCom 2004, Lecture Notes in Computer Science, 2978*, pages 60–75, 2004.

[7] G. Hamon, L. deMoura, and J. Rushby. Generating efficient test sets with a model checker. In *2nd International Conference on Software Engineering and Formal Methods*, pages 261–270. IEEE Computer Society, Sept. 2004.

[8] G. Hamon, L. deMoura, and J. Rushby. Automated test generation with SAL. Technical report, Computer Science Laboratory, SRI, 2005. `http://www.csl.sri.com/users/rushby/abstracts/sal-atg`.

[9] D. Lee, K. K. Sabnani, D. M. Kristol, S. Paul, and M. Ü. Uyar. Conformance testing of protocols specified as communicating fsms. In *INFOCOM*, 1993.

[10] D. Lee and M. Yannakakis. Principles and Methods of Testing Finite State Machines - a Survey. *The Proceedings of IEEE*, 84(8):1089–1123, 1996.

[11] Leonardo de Moura, Sam Owre and N. Shankar. The SAL language manual. Technical report, SRI International, 2003.

[12] P. Miller. Interoperability. what is it and why should i want it? Ariadne Issue 24, June 2000. `http://www.ariadne.ac.uk/issue24/interoperability/intro.html`.

[13] S. Mujjiga and S. Sukumaran. Generating tests for validating interoperability of networked media devices – a formal approach, realisation, and initial results. Philips Research Technical Note 2007-00240, 2007.

[14] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages

267–276, 2003. See
`http://bogor.projects.cis.ksu.edu/`.

[15] S. Seol, M. Kim, and S. T. Chanson. Interoperability test generation for communication protocols based on multiple stimuli principle. In *TestCom*, pages 151–168, 2002.

[16] S. Seol, M. Kim, S. Kang, and J. Ryu. Fully automated interoperability test suite derivation for communication protocols. *Computer Networks*, 43:735–759, 2003.

[17] S. Sukumaran, A. Sreenivas, and R. Venkatesh. A rigorous approach to requirements validation. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*, 2006.

[18] V. Trenkaev, M. Kim, and S. Seol. Interoperability Testing Based on a Fault Model for a System of Communicating FSMs. In *TestCom 2003, LNCS, 2644*, 2003.

[19] T. Walter and B. Plattner. Conformance and interoperability - a critical assessment. Technical Report 9, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology, Zurich, 1994.

Extended Interface Grammars for Automated Stub Generation*

Graham Hughes and Tevfik Bultan
Computer Science Department
University of California
Santa Barbara, CA 93106, USA
{graham,bultan}@cs.ucsb.edu

ABSTRACT

An important challenge in software verification is the ability to verify different software components in isolation. Achieving modularity in software verification requires development of innovative interface specification languages. In this paper we focus on the idea of using grammars for specification of component interfaces. In our earlier work, we investigated characterizing method call sequences using context free grammars. Here, we extend this approach by adding support for specification of complex data structures. An interface grammar for a component specifies the sequences of method invocations that are allowed by that component. Our current extension provides support for specification of valid input arguments and return values in such sequences. Given an interface grammar for a component, our interface compiler automatically generates a stub for that component that 1) checks the ordering of the method calls to that component, 2) checks that the input arguments are valid, and 3) generates appropriate return values based on the interface grammar specification. These automatically generated stubs can be used for modular verification and/or testing. We demonstrate the feasibility of this approach by experimenting with the Java Path Finder (JPF) using the stubs generated by our interface compiler.

1. INTRODUCTION

Modularity is key for scalability of almost all verification and testing techniques. In order to achieve modularity, one has to isolate different components of a program during verification or testing. This requires replacement of different components in a program with stubs that represent their behavior. Our work on interface grammars originates from the following observation: If we can develop sufficiently rich interface specification languages, it should be possible to automatically generate stubs from these rich interfaces, enabling modular verification and testing.

*This work is supported by NSF grant CCF-0614002.

AFM'07, November 6, Atlanta, GA, USA.

In a recent paper [11] we proposed interface grammars as an interface specification language. An interface grammar for a component specifies the sequences of method invocations that are allowed by that component. Using interface grammars one can specify nested call sequences that cannot be specified using interface specification formalisms that rely on finite state machines. We built an interface compiler that takes the interface grammar for a component as input and generates a stub for that component. The resulting stub is a table-driven parser generated from the input interface grammar. Invocation of a method within the component becomes the lookahead symbol for the stub/parser. The stub/parser uses a parser stack, the lookahead, and a parse table to guide the parsing. The interface grammar language proposed in [11] also supports specification of semantic predicates and actions, which are Java code segments that can be used to express additional interface constraints. The semantic predicates and semantic actions that appear in the right hand sides of the production rules are executed when they appear at the top of the stack.

Although the interface grammar language proposed in [11] provides support for specification of allowable call sequences for a component, it does not directly support constraints on the input and output objects that are passed to the component methods as arguments or returned by the component methods as return values. In this paper we investigate the idea of using grammar production rules for expressing constraints on object validation and creation. Our approach builds on shape types [10], a formalism based on graph grammars, which can be used for specification of complex data structures. We show that grammar productions used in shape types can be easily integrated with grammar productions in interface grammars. In order to achieve this integration we allow nonterminals in interface grammars to have arguments that correspond to objects. The resulting interface specification language is capable of expressing constraints on call sequences, as well as constraints on input and output data that is received and generated by the component.

Our work is significantly different from earlier work on interface specification. Most of the earlier work on interfaces focuses on interface specification formalisms based on finite state machines [5, 4, 20, 2, 3]. More expressive interface specification approaches such as the ones based on design by contract [12, 8] are less amenable to automation. Moreover, it is not easy to express control flow related constraints,

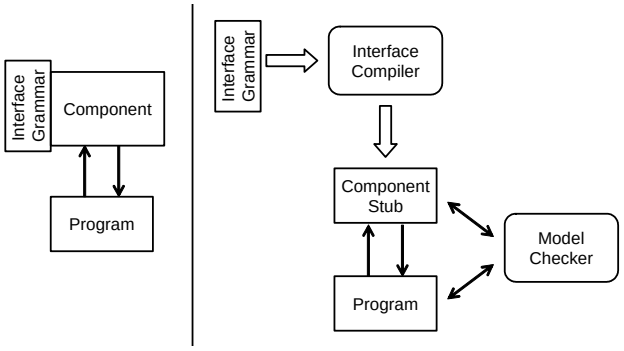


Figure 1: Modular verification with interface grammars.

such as the ones relating to call sequencing, as pre and post-conditions. We believe that the extended interface grammar specification language presented in this paper provides a unique balance between automation and expressiveness, and also enables specification of control flow and data structure constraints in a uniform manner.

There has been earlier work in grammar based testing such as [6, 13, 14, 15]. The common goal in these papers is to automatically generate test inputs using a grammar that characterizes the set of possible inputs. In contrast, in our work we use grammars as interface specifications where terminals correspond to method calls.

The rest of the paper is organized as follows. Section 2 gives an overview of interface grammars and shape types using a tree component as a running example. Section 3 includes a more detailed discussion on shape types. Section 4 discusses integration of concepts from shape types to interface grammars and how they can be used in object generation. Section 5 discusses and contrasts object validation versus object generation. Section 6 presents experiments demonstrating the use of interface grammars for modular verification of EJB clients using the Java PathFinder (JPF) [16] model checker. Section 7 concludes the paper.

2. EXTENDING INTERFACE GRAMMARS

The modular verification approach based on interface grammars is shown in Figure 1. Interface grammars provide a language for specification of component interfaces. The core of an interface grammar is a set of production rules that define a Context Free Grammar (CFG). This CFG specifies all acceptable method call sequences for the given component. Given an interface specification for a component, our interface compiler generates a stub for that component. This stub is a table-driven top-down parser [1] that parses the sequence of incoming method calls (i.e., the method invocations) based on the CFG defined by the interface specification.

As an example, consider a general tree with first-child and right-sibling pointers; for example, an XML Domain Object Model tree. If we add redundant left-sibling and parent pointers to the tree, we can then write a tree cursor that can traverse the tree, with the methods `moveup`, `movedown`, `moveleft` and `moveright`. We may wish to examine traversal algorithms independent of any particular tree representation, and thus want a component for this cursor. We can represent the cursor navigation operations using the following simplified interface grammar:

$$
\begin{array}{lcl}
\mathit{TreeCall} & \rightarrow & \texttt{movedown}\ \mathit{TreeCall} \\
& | & \texttt{movedown}\ \mathit{TreeCall}\ \texttt{moveup}\ \mathit{TreeCall} \\
& | & \texttt{moveright}\ \mathit{TreeCall} \\
& | & \texttt{moveright}\ \mathit{TreeCall}\ \texttt{moveleft}\ \mathit{TreeCall} \\
& | & \epsilon
\end{array}
$$

This is a context free grammar with the nonterminal symbol *TreeCall* (which is also the start symbol) and terminal symbols `moveup`, `movedown`, `moveright`, and `moveleft`. Each terminal symbol corresponds to a method call, and the above grammar describes the allowable call sequences that are supported by the component. We have restricted the permissible call sequences as follows: it is always an error to have more `moveup` symbols than `movedown` (corresponding to trying to take the parent of the root which may result in dereferencing a null pointer) and it is always an error to have more `moveleft` symbols than `moveright` at any given height (corresponding to trying to take the left sibling of the first child which may again result in dereferencing a null pointer). In our framework, this language corresponds to the set of acceptable incoming call sequences for a component, i.e., the interface of the component. Note that the set of acceptable incoming call sequences for the above example cannot be recognized by a finite state machine since the matching of `movedown` and `moveup` symbols, and `moveleft` and `moveright` symbols cannot be done using a finite state machine. The expressive power of a context free grammar is necessary to specify such interfaces. One could also investigate using extended finite state machines to specify such interfaces. However, we believe that grammars provide a suitable and intuitive mechanism for writing interface specifications.

Given the above grammar we can construct a parser which can serve as a stub for the Tree component. This stub/parser will simply use each incoming method call as a lookahead symbol and implement a table driven parsing algorithm. If at some point during the program execution the stub/parser cannot continue parsing, then we know that we have caught an interface violation. In [11] we described such an interface grammar compiler that, given an interface grammar for a component, automatically constructs a stub/parser for that component.

Now, assume that, for the above tree example, we would also like to specify a `getTree` method that returns the tree that is being traversed. This is a query method and it can be called at any point during execution, i.e., there is no restriction on the execution of the `getTree` method as far as the control flow is concerned. However, the return value of the `getTree` method is a specific data structure. It would be helpful to provide support for specification of such data structures at the interface level. Our goal in this paper is to extend our interface grammar specification language to provide support for specification of such constraints. Such constraints can be used to specify the structure of the objects that are passed to a component or returned back from that component.

Consider the interface grammar below which is augmented

by a set of recursive rules that specify the structure of the tree that `getTree` method returns:

```
1   TreeCall      →  movedown TreeCall
2                 |  movedown TreeCall moveup TreeCall
3                 |  moveright TreeCall
4                 |  moveright TreeCall moveleft TreeCall
5                 |  getTree TreeGen x TreeCall
6                 |  ϵ
7   TreeGen x     →  N x null
8   N x y         →  leftc x z, parent x y, N z x, L x y
9                 |  leftc x null, parent x y, L x y
10                |  leftc x null, parent x null
11  L x y         →  rights x z, N z y, L z y
12                |  rights x null
```

The productions 1-4 and 6 are the same productions we used in the earlier interface grammar. The production 5 represents the fact that the modified interface grammar also accepts calls to the `getTree` method. The nonterminal *TreeGen* is used to define the shape of the tree that is returned by the `getTree` method. Productions 7-12 define a *shape type* based on the approach proposed by Fradet and le Métayer [10]. Shape types are based on graph grammars and are used for defining shapes of data structures using recursive rules similar to CFGs.

Before we discuss the shape types in more detail in Section 3, we would like to briefly explain the above example and the data structure it defines. The nonterminals *TreeGen*, *N* and *L* used in the production rules 7-12 have arguments that are denoted as *x, y, z*. Arguments *x, y, z* represent the node objects in the data structure. In this example, the data structure is a left-child, right-sibling (LCRS) tree. In this data structure, each node has a link to its leftmost child, its immediate right sibling, and its parent if they exist, otherwise these fields are set to null. The terminal symbols **leftc**, **rights** and **parent** denote the fields that correspond to the left-child, right-sibling, and parent of a node object, respectively. Each production rule expresses some constraints among its arguments in its right hand side, and recursively applies other production rules to express further constraints. For example **parent** *x y* means that the **parent** field of node *x* should point to node *y*. Similarly, **rights** *x* null means that the **rights** field of node *x* should be null.

In Figure 2 we show an example LCRS tree. Let us investigate how this tree can be created based on the production rules 7-12 shown above. Production 7 states that a LCRS tree can be created using one of the production rules for the nonterminal *N* and by substituting the node corresponding to the root of the tree (i.e., node 1 in Figure 2) for the first argument and null for the second argument. Let us pick production 8 for nonterminal *N* and substitute node 1 for *x*, null for *y* and node 2 for *z*. Based on this assignment, the constraints listed in the right hand side of production 8 state that: **leftc** field of node 1 should point to node 2; **parent** field of node 1 should be null; nodes 2 and 1 should satisfy the constraints generated by a production rule for nonterminal *N* where the first argument is set to node 2 and the second argument is set to node 1; and, node 1 and null should satisfy the constraints generated by a production rule for nonterminal *L* where the first argument is set to node 1 and the second argument is set to null. Note that, the first two constraints are satisfied by the tree shown in Figure 2. The last constraint is satisfied by picking the production rule 12, which states that the **rights** field of node 1 should be null, which is again satisfied by the tree shown in Figure 2. Finally, the third constraint recursively triggers another application of the production 8 where we substitute node 2 for *x*, node 1 for *y*, and null for *z*. By recursively applying the productions rules 7-12 this way, one can show that the tree shown in Figure 2 is a valid LCRS tree based on the above shape type specification.

The above example demonstrates that we can use shape types for object validation. Object validation using shape types corresponds to parsing the input object graphs based on the grammar rules in the shape type specification. Note that we can use shape types for object generation in addition to object validation. In order to create object graphs that correspond to a particular shape type we can randomly pick productions and apply them until we eliminate all nonterminals. Resulting object graph will be a valid instance of the corresponding shape type. In fact, in the above example, our motivation was to use the shape type formalism to specify the valid LCRS trees that are returned by the `getTree` method.

We would like to emphasize that, although we will use data-structures such as LCRS tree as running examples in this paper, our goal is not verification of data structure implementations, or clients of data structure libraries. Rather, our goal is to develop a framework that will allow verification of arbitrary software components in isolation. This requires an interface specification mechanism that is capable of specifying the shapes of the objects that are exchanged between components as method arguments or return values. Our claim is that extended interface grammars and our interface compiler provide a mechanism for isolating components which enables modular verification.

In the following sections we will discuss shape types in more detail and discuss how to integrate them to our interface grammar specification language. We will also demonstrate examples of both object validation and generation with our extended interface grammar specification language based on shape types.

We note one weakness of the above interface specification example. According to the above interface grammar specification, the tree that is returned by the `getTree` method may not be consistent with the previous calls to the `moveup`, `movedown`, `moveleft` and `moveright` methods that have been observed. For example, if a client calls the `movedown` method twice followed by two calls to the `moveup` method, then if the next call is `getTree`, the `getTree` method should return a tree of height greater than or equal to two to be consistent with the observed call history. However, the above specification does not enforce such a constraint. The `getTree` method can return any arbitrary LCRS tree at any time. Our interface specification language is capable of specifying this type of constraints (i.e., making sure that the the tree returned by the `getTree` method is consistent with the past call history to the tree component) using semantic predicates and actions.

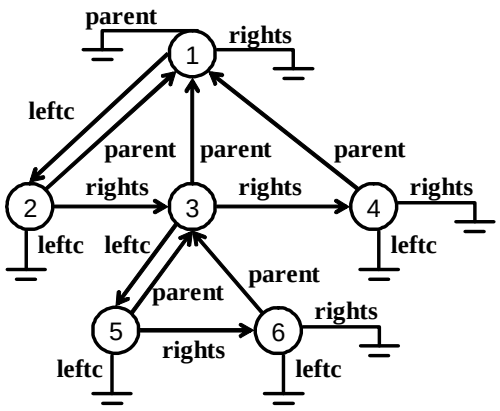


Figure 2: An example left-child, right-sibling (LCRS) tree.

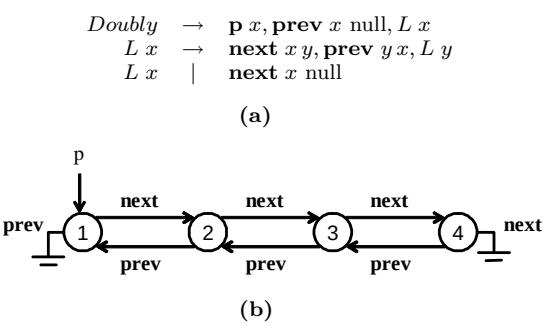


Figure 3: (a) Shape type for a doubly linked list, and (b) an example linked list of that type

3. SHAPE TYPES

A weakness of the interface specification language defined in our previous work [11] is that it does not provide direct support for describing the data associated with the method calls and returns of a component, i.e., the arguments and return values for the component methods. However, the interface specification language presented in [11] allows specification of semantic predicates and actions. This enables the users to insert arbitrary Java code to interface specifications. These semantic predicates and actions can be treated as nonterminals with epsilon-productions and the Java code in them are executed when the corresponding nonterminal appears at the top of the parser stack. The user can do object validation and generation using such semantic predicates and semantic actions. However, this approach is unsatisfactory for the same reason that hand writing a component stub in Java directly is unsatisfactory; it is frequently brittle and difficult to understand. Accordingly, we would like to extend our interface grammars to support generating and validating data, and to do so in a way that preserves the advantages of grammars.

The shape types of Fradet and le Métayer [10] define an attractive formalism based on graph grammars that can be used to express recursive data structures. We have been inspired by their formalism, but to accommodate the differences between their goal and ours our implementation becomes substantially different. Nonetheless, it is worthwhile explaining Fradet and le Métayer's shape types and then explaining how our approach differs syntactically before explaining our implementation.

Shape types are an extension to a traditional type system. Their goal is to extend an underlying type system so that it can specify the shape of a data structure; for example, a doubly linked list. This extension is done through extending a normal context free grammar, which we will proceed to explain.

Consider the language of strings $(\mathbf{name}\ x\ y)^*$, where **name** is some string and x and y are integers. If we regard x and y as vertices, then we can obtain a labeled directed graph from any such string by regarding the string $\mathbf{name}\ x\ y$ as defining an edge from the vertex labeled x to the vertex labeled y, itself labeled **name**. If we further regard the vertices in this graph as representing objects and the edges as representing fields, we can obtain an object graph. Note that this mapping is not 1-1: if the strings are reordered the same graph is obtained.

We can represent external pointers into this object graph by adding strings of the form $\mathbf{p}\ x$; here, the pointer named **p** points to the object x.

We now want a grammar that can output these graph encodings. While we can regard **name** as a terminal, the vertices are not so simple. We extend the context-free grammar to permit parameters; so the production $N\ x\ y \rightarrow \mathbf{next}\ x\ y$ describes the string $\mathbf{next}\ x\ y$, whatever its parameters x and y are. If a variable is referred to in the right hand side of a production but not listed in the parameters, then it represents a new object that has not yet been observed. Fradet and le Métayer use $\mathbf{next}\ x\ x$ to represent terminal links; we prefer to use $\mathbf{next}\ x$ null for the same purpose.

Shape types provide a powerful formalism for specification of object graphs. In Figure 3(a) we show the shape type for a doubly linked list and in Figure 3(b) we show an example doubly linked list of that type. In Figure 4(a) we show the shape type for a binary tree and in Figure 4(b) an example binary tree of that type.

4. OBJECT GENERATION WITH INTERFACE GRAMMARS

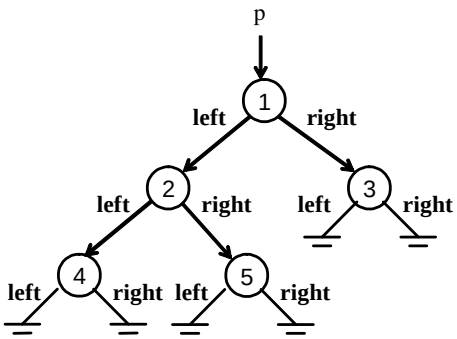
In this section we will discuss how we integrate shape types to our interface grammar specification language. First, we start with a brief discussion on alternative ways of generating arbitrary object graphs in a running Java program. Next, we give an overview of our extended interface grammar language and discuss how this extended language supports shape types. We conclude this section by presenting an example interface grammar for the left-child, right-sibling (LCRS) tree example discussed in Section 2.

4.1 Creating Object Graphs

There are three major techniques for object graph creation: with JVM support, serialization, and method construction. The first technique uses support from the JVM to create

$$
\begin{array}{rcl}
\mathit{Bintree} & \rightarrow & \mathbf{p}\ x, B\ x \\
B\ x & \rightarrow & \mathbf{left}\ x\ y, \mathbf{right}\ x\ z, B\ y, B\ z \\
B\ x & | & \mathbf{left}\ x\ \text{null}, \mathbf{right}\ x\ \text{null}
\end{array}
$$

(a)



(b)

Figure 4: (a) Shape type for a binary tree, and (b) an example binary tree of that type

objects arbitrarily and in any form desired. Visser et al. [18] use this technique, extending the Java PathFinder model checker appropriately. While this is very powerful, it is necessarily coupled to a specific JVM and can be easy to inadvertently create object structures that cannot be recreated by a normal Java program. We reject this approach because we do not want to be overly coupled to a specific JVM.

The second technique uses the Java serialization technologies used by Remote Method Invocation (RMI) [19]. This is almost as powerful as the first technique and has the advantage of being more portable. Since the serialization format is standardized, it is relatively easy to create normal serialization streams by fiat. There are two major issues with this approach. First, it requires that all the objects that one might want to generate be serializable, which requires changing the source code in many cases. Second, it is possible for an object to arbitrarily redefine its serialization format or to add arbitrarily large amounts of extra data to the object stream. This is common in the Java system libraries. Accordingly we have rejected this approach as well.

The third approach, and the one we settled upon, is to generate object graphs through the object's normal methods. The main advantages this has is that it works with any object, it is as portable as the original program, and it is impossible to get an object graph that the program could not itself generate. The main disadvantage is that this approach cannot be fully automated without a specification of the object graph shapes that are valid. Since we do semiautomated analysis, we combine approach with the shape types of the previous section and ask the user to tell us what sort of shapes they desire.

4.2 Extended Interface Grammar Language

In addition to providing support for context free grammar rules, our interface specification language also supports spec-

(1)	*main*	→	*class* *
(2)	*class*	→	`class` CLASSID { *item* * }
(3)	*item*	→	*semact* ;
(4)		|	*rule*
(5)	*rule*	→	`rule` RULEID (*declaration* *) *block*
(6)	*block*	→	{ *statement* * }
(7)	*statement*	→	*block*
(8)		|	`apply` RULEID (ID *) ;
(9)		|	*semact* ;
(10)		|	*declaration* = *semexpr* ;
(11)		|	`choose` { *cbody* * }
(12)		|	? MINVOCATION ;
(13)		|	`return` MRETURN *semexpr* ? ;
(14)		|	`!` MCALL ;
(15)	*cbody*	→	`case` *select* ? : { *statement* * }
(16)	*select*	→	? MINVOCATION *sempred* ?
(17)		|	*sempred*
(18)	*sempred*	→	⟪ EXPR ⟫
(19)	*semexpr*	→	⟪ EXPR ⟫
(20)	*semact*	→	⟪ STATEMENT ⟫
(21)	*declaration*	→	TYPE ID

Figure 5: Abstract syntax for the extended interface grammar language

ification semantic predicates and semantic actions that can be used to write complex interface constraints. A semantic predicate is a piece of code that can influence the parse, whereas a semantic action is a piece of code that is executed during the parse. Semantic predicates and actions provide a way to escape out of the CFG framework and write Java code that becomes part of the component stub. The semantic predicates and actions are inserted to the right hand sides of the production rules, and they are executed at the appropriate time during the program execution (i.e., when the parser finds them at the top of the parse stack).

In Figure 5 we show a (simplified) grammar defining the abstract syntax of our interface grammar language. We denote *nonterminal* and `terminal` symbols and Java CODE and IDENTIFIERS with different fonts. The symbols ⟪ and ⟫ are used to enclose Java statements and expressions. Incoming method calls to the component (i.e., method invocations) are shown with adding the symbol ? to the method name as a prefix. Outgoing method calls (i.e., method calls by the component) are shown with adding the symbol `!` to the method name as a prefix. In the grammar shown in Figure 5, we use "*" to denote zero or more repetitions of the preceding symbol, and "?" to denote that the preceding symbol can appear zero or one times.

An interface grammar consists of a set of class interfaces (not to be confused with Java interfaces) (represented in rule (1) in Figure 5). The interface compiler generates one stub class

for each class interface. Each class interface consists of a set of semantic actions and a set of production rules that define the CFG for that class (rules (2), (3) and (4)). A *semantic action* is simply a piece of Java code that is inserted to the stub class that is generated for the component (rule (20)). A *rule* corresponds to a production rule in the interface grammar. Each rule has a name, a list of declarations, and a block (rule (5)). The use of declarations will be explained in Section 4. A rule block consists of a sequence of statements (rule (6)). Each statement can be a rule application, a semantic action, a declaration, a choose block, a method invocation, a method return or a method call (rules (7)-(14)). A semantic action corresponds to a piece of Java code that is executed when the parser sees the nonterminal that corresponds to that semantic action at the top of the parse stack. A *rule application* corresponds to the case where a nonterminal appears on the right hand side of a production rule. A *declaration* corresponds to a Java code block where a variable is declared and is assigned a value (rule (21)). A *choose block* is simply a switch statement (rules (11) and (15)). A selector for a switch case can either be a method invocation (i.e., an incoming method call), a semantic predicate or the combination of both (rules (16) and (17)). A switch case is selected if the semantic predicate is true and if the lookahead token matches to the method invocation for that switch case. A *method return* simply corresponds to a return statement in Java. When the component stub receives a method invocation from the program, it first calls the interface parser with the incoming method invocation, which is the lookahaed token for the interface parser. When the parser returns, the component stub calls the interface parser again, this time with the token which corresponds to the method return. Finally, a *method call* is simply a call to another method by the stub.

4.3 Support for Shape Types

We can obtain all the power required to embed the shape types of Section 3 into our interface grammars with the following addition: we permit rules to have parameters. Because we need to be able to pass objects to the rules as well as retrieve them, we have chosen to use call-by-value-return semantics for our parameters rather like the "in out" parameters of the Ada language. These parameters are reflected in the declaration list of line 5 of Figure 5, and in the identifier list of line 8 of that same figure. Because we have chosen uniform call-by-value-return semantics, only variable names may be supplied to `apply`.

Because our previous work required lexical scoping, the runtime needed only to be changed as follows: when encoding an `apply`, store the current contents of all its variables in a special location—we currently assign parameter n to variable $-(n+1)$, as all our variables have a nonnegative associated integer used in scoping—push the nonterminal onto the stack as normal, and afterward overwrite each variable with the result, again stored in the special location. That is, if $\langle\!\langle x \rangle\!\rangle$ is the closure performing x and $a_0, \ldots a_n$ is the list of arguments, then the series of grammar tokens corresponding to `apply` *rule* $(a_0, \ldots a_n)$ is

for $i = 0$ **to** n **do**
  $\langle\!\langle$`symbols.put ($(-(i+1)),`
    `symbols.get ($(`a_i`.id)));`$\rangle\!\rangle$
od
rule
for $i = 0$ **to** n **do**
  $\langle\!\langle$`symbols.put ($(`a_i`.id),`
    `symbols.get ($(-(`$i+1$`))));`$\rangle\!\rangle$
od

Similarly, for every production for a `rule`, the compiler must, at the start of the production, bind all its parameters from the special location; and at the end it must store the current values of each of its parameters to the appropriate place in the special location. For example, given a production $A\ a_0 \ldots a_n \rightarrow x_0 \ldots x_m$, the amended production would be as follows:

$A \rightarrow$ ($\langle\!\langle$`symbols.push ();`$\rangle\!\rangle$,
  for $i = 0$ **to** n **do**
    $\langle\!\langle$`symbols.bind ($(`a_i`.id));`$\rangle\!\rangle$,
    $\langle\!\langle$`symbols.put ($(`a_i`.id),`
      `symbols.get ($(-(`$i+1$`))));`$\rangle\!\rangle$
  od,
  $x_0, \ldots x_m$,
  for $i = 0$ **to** n **do**
    $\langle\!\langle$`symbols.put ($(-(`$i+1$`)),`
      `symbols.get ($(`a_i`.id)));`$\rangle\!\rangle$
  od,
  $\langle\!\langle$`symbols.pop ();`$\rangle\!\rangle$)

5. OBJECT GENERATION VS. OBJECT VALIDATION

Using the extended interface grammar specification language presented in Section 4 it is possible to specify both generation and validation of data structures, and to do so in a manner that is reminiscent of the shape types of Section 3. Object validation is used to check that the arguments passed to a component by its clients satisfy the constraints specified by the component interface. Object generation, on the other hand, is used to create the objects that are returned by the component methods based on the constraints specified in the component interface.

Figure 6, shows object generation and validation for doubly linked list and binary tree examples. Figure 6 contains three specifications for each of the two examples. At the top of the figure we repeat the shape type specifications for doubly linked list and binary tree examples from Section 3 for convenience. The middle of the figure contains the interface grammar rules for generation of these data structures. Note the close similarity between the shape type productions and the productions in the interface grammar specification. The bottom of the figure shows the interface grammar rules for validation of these data structures.

Object generation and validation tasks are broadly symmetric, and their specification as interface grammar rules reflects this symmetry as seen in Figure 6. While in object generation semantic actions are used to set the fields of objects to appropriate values dictated by the shape type specification, in object validation, these constraints are checked using semantic predicates specified as guards. Note that the set of nonterminals and productions used for object generation and validation are the same.

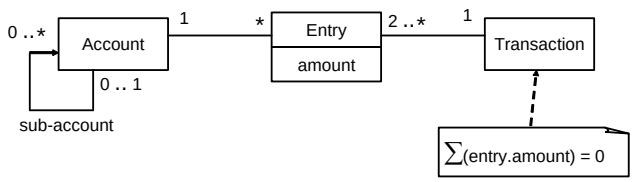


Figure 7: UML diagram of the Account pattern

The most significant difference between the object generation and validation tasks is the treatment of aliasing among different nodes in an object graph. The semantics of the shape type formalism makes some implicit assumptions about aliasing between the nodes. Intuitively, shape type formalism assumes that there is no aliasing among the nodes of the object graph unless it is explicitly stated. During object generation it is easy to maintain this assumption. During generation, every `new` statement creates a new object what is not shared with any other object in the system. If the specified data structure requires aliasing, this can be achieved by passing nodes as arguments as is done in shape type formalism.

Detecting aliasing among objects is necessary during object validation. Note that, since shape type formalism assumes that no aliasing should occur unless it is explicitly specified, during object validation we need to make sure that there is no unspecified aliasing. Instead of trying to enforce a fixed policy on aliasing, we leave the specification of the aliasing policy during object validation to the user. The typical way to check aliasing would be by using a hash-set as demonstrated by the two object validation examples shown in Figure 6. Note that, the interface grammar rules for object validation propagate the set of nodes that have been observed and make sure that there is no unspecified aliasing among them.

6. VERIFICATION WITH INTERFACE GRAMMARS

In this Section, we report some experiments on modular verification of Java programs using stubs automatically generated by our interface compiler. We use the model checker Java PathFinder (JPF) [17] as our verification tool. JPF is an explicit and finite state model checker that works directly on Java bytecode. It enables the verification of arbitrary pure Java implementations without any restrictions on data types. JPF supports property specifications via assertions that are embedded into the source code. It exhaustively traverses all possible execution paths for assertion violations. If JPF finds an assertion violation during verification, it produces a counter-example which is a program trace leading to that violation.

To analyze the performance of stubs automatically generated by our interface compiler, we have written several small clients for an Enterprise Java Beans [7] (EJB) persistence layer. We used a similar technique in our prior work [11]; here we handle some types of queries and perform relational integrity checks upon the resulting database.

We have chosen to base our clients around the Account pattern from Fowler [9]. Strictly speaking this is a pattern for an object schema; accordingly we have implemented it for these tests with the SQL mapping in the EJB framework. The Account pattern is useful for us because it represents structured data and also has a hierarchical element (accounts can have sub-accounts).

A brief description of the Account pattern and how we interpreted it is in order. A UML diagram illustrating all this can be seen in Figure 7. An *account* contains entries and can be a parent to other accounts; the account instances make up a forest. An *entry* is associated with exactly one account and exactly one monetary transaction, and has a field representing an amount of money. A *monetary transaction* is associated with at least two entries, and the sum of all entries in every monetary transaction must be zero at the end of a database transaction—this is often stated as "money is neither created nor destroyed." Since unfortunately the term 'transaction' here refers to two distinct concepts both of which are important to us, we must be explicit: in the absence qualification, 'transaction' always refers to a monetary transaction.

This structure possesses a number of natural invariants. We have already mentioned the key transaction invariant. Accounts and their children must possess the tree property; that is an account can not have two parents. The sum of all entries in all accounts in the system should also be zero; if it is not, we may have forgotten to store an account, an entry, or a transaction. Because we permit more than only two entries per transaction, our transactions are called multi-legged; it is usually considered undesirable or an outright error for one transaction to have more than one "leg" in any one account.

All these data invariants are in addition to the order in which the methods should be called. No query parameters should be adjusted following execution of the query. The queries themselves ought to be executed during a database transaction in order to obtain a consistent view of the database between each query. The *getResultList* or *getSingleResult* methods should be the last operation performed on the query object—these methods request either all results from a single database query or only one result.

We have used our interface grammar compiler to create a stub for the EJB Persistence API that encodes all these invariants. Because the database can change in unpredictable and arbitrary ways between database transactions, our stub entirely regenerates the database every time a transaction is begun. If a transaction is rolled back, it could well be in an incomplete state and so applying database invariants is folly; yet if a transaction is committed it must be verified.

Our stub contains two tunable parameters, corresponding to an upper bound on the number of accounts in the system and an upper bound on the number of entries in the system. The number of transactions in the system is always nondeterministically chosen to be between 1 and $\lfloor\, |\, entries\, |\, /2 \,\rfloor$, inclusive.

To exercise this stub, we have written four EJB Persistence

Shape Type Specification

$$
\begin{array}{rcl}
\mathit{Doubly} & \rightarrow & \mathbf{p}\ x, \mathbf{prev}\ x\ \mathrm{null}, L\ x \\
L\ x & \rightarrow & \mathbf{next}\ x\ y, \mathbf{prev}\ y\ x, L\ y \\
L\ x & \mid & \mathbf{next}\ x\ \mathrm{null}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Bintree} & \rightarrow & \mathbf{p}\ x, B\ x \\
B\ x & \rightarrow & \mathbf{left}\ x\ y, \mathbf{right}\ x\ z, B\ y, B\ z \\
B\ x & \mid & \mathbf{left}\ x\ \mathrm{null}, \mathbf{right}\ x\ \mathrm{null}
\end{array}
$$

Object Generation with Interface Grammars

```
rule genDoubly (Node x) {
  ⟪ x = new Node (); ⟫
  ⟪ x.setPrev (null); ⟫
  apply genL (x);
}
rule genL (Node x) {
  choose {
  case:
    Node y = ⟪ new Node () ⟫;
    ⟪ x.setNext (y); ⟫
    ⟪ y.setPrev (x); ⟫
    apply genL (y);
  case:
    ⟪ x.setNext (null); ⟫
  }
}
```

```
rule genBintree (Node x) {
  ⟪ x = new Node (); ⟫
  apply genB (x);
}
rule genB (Node x) {
  choose {
  case:
    Node y = ⟪ new Node (); ⟫;
    Node z = ⟪ new Node (); ⟫;
    ⟪ x.setLeft (y); ⟫
    ⟪ x.setRight (z); ⟫
    apply genB (y);
    apply genB (z);
  case:
    ⟪ x.setLeft (null); ⟫
    ⟪ x.setRight (null); ⟫
  }
}
```

Object Validation with Interface Grammars

```
rule matchDoubly (Node x) {
  Set nodesSeen = ⟪ new HashSet () ⟫;
  guard ⟪ x instanceof Node
    && !nodesSeen.contains (x) ⟫;
  ⟪ nodesSeen.insert (x); ⟫
  guard ⟪ x.getPrev () == null ⟫;
  apply matchL (x, nodesSeen);
}
rule matchL (Node x, Set nodesSeen) {
  choose {
  case ⟪ x.getNext () == null ⟫:
  case ⟪ x.getNext () != null ⟫:
    Node y = ⟪ x.getNext () ⟫;
    guard ⟪ y instanceof Node
      && !nodesSeen.contains (y) ⟫;
    ⟪ nodesSeen.insert (y); ⟫
    guard ⟪ x.getNext () == y ⟫;
    guard ⟪ y.getPrev () == x ⟫;
    apply matchL (y, nodesSeen);
  }
}
```

```
rule matchBintree (Node x) {
  Set nodesSeen = ⟪ new HashSet () ⟫;
  guard ⟪ x instanceof Node
    && !nodesSeen.contains (x) ⟫;
  ⟪ nodesSeen.insert (x); ⟫
  apply matchB (x, nodesSeen);
}
rule matchB (Node x, Set nodesSeen) {
  choose {
  case ⟪ x.getLeft () == null ⟫:
    guard ⟪ x.getRight () == null ⟫;
  case ⟪ x.getLeft () != null ⟫:
    Node y = ⟪ x.getLeft () ⟫;
    guard ⟪ y instanceof Node
      && !nodesSeen.contains (y) ⟫;
    ⟪ nodesSeen.insert (y); ⟫
    Node z = ⟪ x.getRight () ⟫;
    guard ⟪ z instanceof Node
      && !nodesSeen.contains (z) ⟫;
    ⟪ nodesSeen.insert (z); ⟫
    guard ⟪ x.getLeft () == y ⟫;
    guard ⟪ x.getRight () == z ⟫;
    apply matchB (y, nodesSeen);
    apply matchB (z, nodesSeen);
  }
}
```

Figure 6: Interface grammars for doubly linked list and binary tree generation and matching

Correct clients				Incorrect clients					
deparent		*voider*		*reparent*		*increaser*		Accounts	Entries
0:11	26 MB	0:17	27 MB	0:10	27 MB	0:14	27 MB	1	2
0:14	26 MB	0:23	37 MB	0:16	36 MB	0:13	27 MB	1	4
0:21	34 MB	0:38	39 MB	0:20	36 MB	0:14	27 MB	1	6
0:49	36 MB	2:55	41 MB	0:17	36 MB	0:14	27 MB	1	8
3:38	36 MB	15:37	50 MB	0:18	36 MB	0:14	27 MB	1	10

Table 1: Run time and memory usage vs. number of entries

Correct clients				Incorrect clients					
deparent		*voider*		*reparent*		*increaser*		Accounts	Entries
0:14	26 MB	0:23	37 MB	0:16	36 MB	0:13	27 MB	1	4
1:09	35 MB	2:35	41 MB	0:56	38 MB	0:13	27 MB	2	4
19:09	37 MB	34:18	43 MB	14:03	39 MB	0:19	27 MB	3	4

Table 2: Run time and memory usage vs. number of accounts

API clients, and have run the clients with varying parameters in the JPF model checker. Two clients are correct in their use of the database and we expect that JPF will report this. Two are incorrect; one triggers a fault almost immediately, and the other is only invalid some of the time. Our clients are as follows:

1. *deparent* takes an account and removes it from its parent.

2. *voider* selects a transaction and 'voids' it, by creating a new transaction negating the original transaction. This introduces new objects into the system.

3. *reparent* takes two entries in the system and trades their transactions.

4. *increaser* increases the monetary value of entries in the system.

Our results are presented in Tables 1 and 2. When there is an interface violation, JPF halts at the first assertion violation and reports an error. In the experiments reported in Table 1 we restricted the state space to a single account and we observed the change in the verification results with respect to increasing number of entries. In the experiments reported in Table 2 we restricted the number of entries and increased the number of accounts.

The *deparent* client removes parent of an account. Changing the parent of an account can cause cycles if done naïvely, but removing the parent is always safe. Since the *deparent* client does not violate any interface properties, JPF does not report any assertion violations for *deparent*. As the number of accounts and entries increases verifying this operation takes an exponentially increasing amount of time to complete due to exponential increase in the state space.

The *voider* client does not violate any interface properties and, hence, JPF does not report any assertions violations for *voider*. Since *voider* introduces new objects to the system it creates a larger state space and its verification takes a longer time than *deparent*.

The *reparent* swaps the transactions of two entries. If the entries encode the same monetary value this can be safe, but in the general case this operation will break the transaction invariant. An additional complication is that if there are less than four entries in the system, *reparent* cannot fail; there is only one transaction available. The time it takes for model checker to reach an assertion violation for *reparent* depends on the order the model checker explores the states.

However the proportion of the state space where *reparent* is valid decreases precipitously as the number of entries in the system increases. Accordingly we expect that the running time will eventually come to some equilibrium if we increase the number of entries, but will consume an exponentially increasing amount of time if we hold the number of entries constant and increase the number of accounts (as observed in Table 2.

Since the *increaser* client always increases the monetary values of the entries, it always violates the transaction invariant, with even two entries in the system. So it takes model checker approximately the same amount of time to report an assertion violation for the *increaser* client regardless of the size of the state space.

7. CONCLUSION

We presented an extension to interface grammars that supports object validation and object creation. The presented extension enables specification of complex data structures such as trees and linked lists using recursive grammar rules. The extended interface grammar specification language provides a uniform approach for specification of allowable call sequences and allowable input and output data for a component. Given the interface grammar for a component, our interface compiler automatically creates a stub for that component which can be used for modular verification or testing. We demonstrated the use of interface grammars for modular verification by conducting experiments with JPF using stubs automatically generated by our interface compiler.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Prog. Languages, (POPL 2005)*, 2005.

[3] A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 248–257, 2004.

[4] A. Chakrabarti, L. de Alfaro, T. Henzinger, M. Jurdziński, and F. Mang. Interface compatibility checking for software modules. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 428–441, 2002.

[5] L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings 9th Annual Symposium on Foundations of Software Engineering*, pages 109–120, 2001.

[6] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 170–178, 1981.

[7] Enterprise java beans 3.0 specification. Technical report, Sun Java Community Process, May 2006. JSR-000220.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2002)*, pages 234–245, 2002.

[9] M. Fowler. *Analysis Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.

[10] P. Fradet and D. le Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997. ACM Press.

[11] G. Hughes and T. Bultan. Interface grammars for modular software model checking. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, 2007. To appear.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, March 2006.

[13] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, 1990.

[14] P. M. Maurer. The design and implementation of a grammar-based data generator. *Softw., Pract. Exper.*, 22(3):223–244, 1992.

[15] E. G. Sirer and B. N. Bershad. Using production grammars in software testing. In *Proceedings of the 2nd Conference on Domain-Specific Languages (DSL 99)*, pages 1–13, 1999.

[16] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the The Fifteenth IEEE International Conference on Automated Software Engineering (ASE'00)*, page 3. IEEE Computer Society, 2000.

[17] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.

[18] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with java pathfinder. In *Proceedings of International Symp. on Software Testing*, 2004.

[19] J. Waldo. Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7, July–September 1998.

[20] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the 2002 ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, 2002.

Cooperative Reasoning for Automatic Software Verification

Andrew Ireland
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh
a.ireland@hw.ac.uk

Position Paper

ABSTRACT

Separation logic was designed to simplify pointer program proofs. In terms of verification tools, the majority of effort has gone into developing light-weight analysis techniques for separation logic, such as shape analysis. Shape analysis ignores the content of data, focusing instead on how data is structured. While such light-weight properties can be extremely valuable, ultimately a more comprehensive level of specification is called for, *i.e.* correctness specifications. However, to verify such comprehensive specifications requires more heavy-weight analysis, *i.e.* theorem proving. We propose an integrated approach for the automatic verification of correctness specifications within separation logic. An approach which combines both light-weight and heavy-weight techniques is proposed. We are aiming for a cooperative style of integration, in which individual techniques combine their strengths, but crucially compensate for each other's weaknesses through the communication of partial results and failures.

1. INTRODUCTION

The proliferation of software across all aspects of modern life means that software failures can have significant economic, as well as social impact. The goal of being able to develop software that can be formally verified as correct with respect to its intended behaviour is therefore highly desirable. The foundations of such formal verification have a long and distinguished history, dating back over fifty years [12, 13, 15]. What has remained more elusive are *scalable verification tools* that can deal with the complexities of software systems.

However, times are changing, as reflected by a current renaissance within the formal software verification community. An IFIP working group has been set up with the aim of developing a *Grand Challenge* for *Verified Software* [23, 28]. There have also been some notable industrial success stories. For instance, Microsoft's Static Device Verifier (SDV) [29] and the SPARK Approach to developing high integrity software [1]. Both of these successes address the scalability issue by focusing on generic properties and tool integrations that support a high degree of automation. In the case of SDV, the focus is on *deadlock freedom* at the level of resource ownership for device driver software. Abstraction and model checking are used to identify potential defects, which are then refined via theorem proving to eliminate false alarms. The SPARK Approach has been used extensively in the development of safety [24] and security [14] critical applications. It provides a loose coupling of analysis techniques from data and information flow analysis through to formal verification via theorem proving. One of its key selling points is its support for automating so called *exception freedom* proofs, *i.e.* proving that a system is free from common run-time errors such as buffer overflows.

The targeting of generic properties, such as deadlock and exception freedom, has proved both highly effective and extremely valuable to industry. However, to increase the value of software correctness guarantees will ultimately call for a more comprehensive level of specification, *i.e.* correctness specifications. In the case of bespoke applications, this might take the form of correctness specifications developed in conjunction with the customer requirements. Alternatively, the verification of software libraries and components against agreed correctness standards could prove highly valuable across a wide range of sectors. Verifying code against more comprehensive specifications will call for significant advances in terms of scalable tools. We believe that these advances will require frameworks which provide, i) general support for *modular reasoning* as well as, ii) a flexible basis in which novel *tool integrations* can be investigated. Here we outline a proposal which builds upon *separation logic*, where modular reasoning is a key feature. At the level of tool integration, we propose the use of *proof planning*, an automated theorem proving technique which has a track-record in successfully combining reasoning tools.

2. SEPARATION LOGIC

Separation logic was developed as an extension to Hoare logic [15], with the aim of simplifying pointer program verification proofs [32, 35]. Pointers are a powerful and widely used programming mechanism, but developing and maintaining correct pointer programs is notoriously hard. A key feature of separation logic is that it focuses the reasoning

AFM'07, November 6, Atlanta, GA, USA.

effort on only those parts of the heap that are relevant to a program, so called *local reasoning*. Because it deals smoothly with pointers, including "dirty" features such as memory disposal and address arithmetic, separation logic holds the promise of allowing verification technology to be applied to a much wider range of **real-world software** than has been possible up to now.

In terms of tool development, the main focus has been on shape analysis. Such analysis can be used to verify properties about the structure (*shape*) of data structures within the heap. For example, given a sorting program that operates on singly linked list, then shape analysis techniques can be used to verify that for an arbitrary singly linked list the program will always output a singly linked list. Note that shape analysis ignores the *content* of data structures. Smallfoot [3] is an experimental tool that supports the automatic verification of shape properties specified in separation logic. Smallfoot uses a form of symbolic execution [4], where loop invariants are required. Related tools are SLAyer [2], Space Invader [10] and Smallfoot-RG [9], all of which build directly upon the foundations of Smallfoot. Within SLAyer, higher-order generic predicates are used to express families of complex composite data structures. A restricted form of predicate synthesis is used to instantiate the generic predicates during shape analysis. Space Invader, unlike Smallfoot supports loop invariant discovery via fixed point analysis. Abstraction is used to overcome divergence in the search for a fixed point. Smallfoot-RG also includes Space Invader's invariant discovery strategy. Closely related to Space Invader is an algorithm developed at CMU for inferring loop invariants within the context of separation logic [25]; again fixed point analysis is the underlying mechanism. Another interesting tool is reported in [30], which combines shape and size analysis. Inductive predicates play a significant role in specifying pointer programs within separation logic. A limitation of the program analysis tools mentioned above is that they are hard-wired with pre-defined inductive predicates, *e.g.* singly linked lists, binary trees etc. To make these tools extensible would require the ability to add new user-defined inductive predicates on-the-fly. To achieve this would require the ability to automate proof by mathematical induction. We will return to this point later.

Less work has been undertaken in the area of theorem proving for separation logic. In [34] a partial formalization within PVS [33] is presented which supports the verification of recursive procedures. A complementary formalization, which includes simple while loops, but not recursive procedures, is presented in [39]. In [38] a shallow embedding of separation logic within Isabelle/HOL [31] is presented, building upon Schirmer's verification environment for sequential imperative programs [36]. This integration was used to reason about pointer programs written in C. Finally, in [26] the Coq proof environment has been extended with separation logic in order to verify the C source code of the Topsy heap manager. All these applications of theorem proving to separation logic have involved significant user interaction, *e.g.* user specified induction rules and loop invariants. In contrast, our proposal focuses on **verification automation** in which user interaction is eliminated as far as possible.

3. PROOF PLANNING

Proof planning is a technique for automating the search for proofs through the use of high-level proof outlines, known as *proof plans* [5]. The current state-of-the-art proof planner is called IsaPlanner [11], which is Isabelle based. Proof planning has been used extensively for proof by mathematical induction [8]. Mathematical induction is essential for the synthesis and verification of the inductively defined predicates that arise within separation logic specifications. Proof planning therefore offers significant benefits for reasoning about separation logic specifications. In addition, the kinds of data structures that arise naturally when reasoning about pointer programs, *i.e.* a queue implemented as a "circular" linked list, will provide challenging examples which will advance the existing proof plans. A distinctive feature of proof planning is *middle-out reasoning* [7], a technique where meta-variables, typically higher-order, are used to delay choice during the search for a proof. Middle-out reasoning has been used to greatest effect within the context of *proof critics* [16], a technique that supports the automatic analysis and patching of failed proof attempts. Such *proof patching* has been applied successfully to the problems of inductive conjecture generalization and lemma discovery [19, 20], as well as loop invariant discovery [22]. This work is currently being integrated and extended within IsaPlanner. The tool integration capabilities of proof planning have been demonstrated through the Clam-HOL [37] and NuSPADE projects[1] [21]. The NuSPADE project targeted the SPARK Approach [1], and integrated proof planning with light-weight program analysis in order to increase proof automation for loop-based code. The resulting integration was applied to industrial strength problems and successfully increased the level of proof automation for exception freedom proofs [21].

4. PROPOSED COOPERATION

In terms of tool integration, we are interested in *tight* integrations where their is *real cooperation* between complementary techniques. That is, where individual techniques combine their strengths, but crucially compensate for each other's weaknesses through the communication of partial results and failures. More general evidence as to the merits of such cooperation can be found in [6]. For us the pay-off of achieving this level of cooperation will be measured in terms of automation, *i.e.* we believe that this form of cooperation will deliver verification automation where skilled human interaction is currently essential.

Our starting point is the proof planning paradigm, and the Smallfoot family of program analyzers. We see two areas where a cooperative style of reasoning could make an impact, *i.e.* when reasoning about recursive and iterative code. In the case of recursive code, the central rule of separation logic comes into play, *i.e.* the *frame* rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{R * P\}\ C\ \{R * Q\}} \quad \begin{array}{l}\text{no variable occurring free}\\ \text{in } R \text{ is modified by } C.\end{array}$$

Note that $*$ denotes *separating conjunction*, where $R * Q$ holds for a heap if the heap can be divided into two disjoint heaps H_1 and H_2 where R holds for H_1 and Q holds for H_2. The frame rule under-pins the notion of local reasoning mentioned in §2, and plays a pivotal role in reasoning about

[1]NuSPADE project: `http://www.macs.hw.ac.uk/nuspade`

recursive procedure calls. Note that the invariant R corresponds to what McCarthy and Hayes refer to as the "frame axiom" [27].

Switching from recursion to iteration, the need for frame axioms is replaced by the need for loop invariants. Consider for example the Hoare style proof rule for a while-loop:

$$\frac{\{P \rightarrow R\} \quad \{R \wedge S\}\, C\, \{R\} \quad \{\neg S \wedge R \rightarrow Q\}}{\{P\}\ \textbf{while}\ S\ \textbf{do}\ \{R\}\, C\, \textbf{od}\ \{Q\}}$$

Here R denotes the loop invariant. Note that in both the frame and while-loop rules, R will typically not form part of a program's overall correctness specification. That is, R represents an auxiliary specification, a *eureka step*, typically supplied via user interaction. Automating the discovery of R represents a significant challenge to achieving verification automation. Our proposal directly addresses this challenge. We are focusing on correctness specifications, so R describes both the *shape* and *content* of heap data structures. Note that Smallfoot, and its related program analysis tools, support the automatic discovery of shape properties, but they do not address the issue of content. We believe that proof planning, via middle-out reasoning and proof patching, will enable shape properties to be automatically extended to include properties about the content of data structures within the heap. The Smallfoot tools provide strength in terms of automating the discovery of shape properties while the strength of IsaPlanner lies in its ability to automate the discovery of properties about the content of data structures. In addition, the inductive theorem proving capabilities of IsaPlanner will enable us to compensate for the limitations of current program analysis tools, *i.e.* as mentioned above, extensibility requires the ability to automatically reason about inductively defined predicates. More details on how we believe real cooperation can be achieved are provided in [17, 18]. In terms of systems building, we are currently looking into using Schirmer's generic verification environment mentioned in §2. This will enable us to build upon Tuch's shallow embedding of separation logic [38], as well as IsaPlanner.

5. CONCLUSION

Separation logic is still a relatively new avenue of research, but holds the promise of delivering significant benefits in terms of scalable software verification techniques. To date the majority of research has concentrated on the development of relatively light-weight verification techniques, such as shape analysis. For this reason we believe that our proposal is very timely, as it focuses on combining the results from shape analysis with inductive theorem proving via proof planning. We believe that adopting a cooperative style of integration will enable more comprehensive properties, such as functional correctness, to be addressed.

Acknowledgements: The ideas outlined in this position paper were developed with support from EPSRC Platform grant EP/E005713. Thanks go to Alan Bundy, Ianthe Hind, Paul Jackson, Ewen Maclean, Peter O'Hearn and Alan Smaill for their feedback and encouragement.

6. REFERENCES

[1] J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley, 2003.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. 2007. To appear at CAV'07.

[3] J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[5] A. Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[6] A. Bundy. Cooperating reasoning processes: more than just the sum of their parts. In M. Veloso, editor, *Proceedings of IJCAI 2007*, pages 2–11. IJCAI Inc, 2007. Acceptance speech for Research Excellence Award.

[7] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L. H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[8] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[9] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular safety checking for fine-grained concurrency. In *To appear in the Proceedings of SAS 2007*, 2007.

[10] D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*, volume 3920 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2006.

[11] L. Dixon and J. D. Fleuriot. IsaPlanner: A prototype proof planner in Isabelle. In *Proceedings of CADE'03*, volume 2741 of *LNCS*, pages 279–283, 2003.

[12] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32. American Mathematical Society, 1967.

[13] H.H. Goldstine and J. von Neumann. Planning and coding of problems for an electronic computing instrument. In A.H. Taub, editor, *J. von Neumann: Collected Works*, pages 80–151. Pergamon Press, 1963. Originally, part II, vol. 1 of a report of the U.S. Ordinance Department 1947.

[14] A. Hall and R. Chapman. Correctness by construction: Developing a commercial secure system. *IEEE Software*, 19(2), 2002.

[15] C.A.R. Hoare. An axiomatic basis for computer

programming. *Communications of the ACM*, 12:576–583, 1969.

[16] A. Ireland. The use of planning critics in mechanizing inductive proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning (LPAR'92), St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[17] A. Ireland. Towards automatic assertion refinement for separation logic. In *Proceedings of the* 21^{st} *IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 2006. A longer version is available from the School of Mathematical and Computer Sciences, Heriot-Watt University, as Technical Report HW-MACS-TR-0039: `http://www.macs.hw.ac.uk:8080/techreps/`.

[18] A. Ireland. A cooperative approach to loop invariant discovery for pointer programs. In *Proceedings of 1st International Workshop on Invariant Generation (WING), a satellite workshop of Calculemus 2007, held at RISC, Hagenberg, Austria*, 2007. Appears within the RISC Report Series, University of Linz, Austria, `http://www.risc.uni-linz.ac.at/publications/`.

[19] A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2):79–111, 1996. Also available as DAI Research Paper No 716, Dept. of Artificial Intelligence, Edinburgh.

[20] A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2):225–245, March 1999. A longer version is available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/97/11.

[21] A. Ireland, B. J. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning: Special Issue on Empirically Successful Automated Reasoning*, 36(4):379–410, 2006.

[22] A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1-4):65–97, February 2001. An earlier version is available as Research Memo RM/00/3, Dept. of Computing and Electrical Engineering, Heriot-Watt University.

[23] C.B. Jones, P. O'Hearn, and J. Woodcock. Verified software: A grand challenge. In *IEEE Computer*, 2006.

[24] S. King, J. Hammond, R. Chapman, and A. Pryor. Is proof more cost effective than testing? *IEEE Trans. on SE*, 26(8), 2000.

[25] S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring invariants in separation logic for imperative list-processing programs. In *Proceedings of the Third Workshop on Semantics, Program Analysis and Computing Environments for Memory Management (SPACE'06)*, pages 47–60, Charleston, SC, 2006.

[26] N. Marti, R. Affeldt, and A. Yonezawa. Formal verification of the heap manager of an operating system using separation logic. In *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 400–419. Springer, 2006.

[27] J. McCarthy and P. Hayes. Some philsophical problems from the standpoint of artificial intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*. Edinburgh University Press, 1969.

[28] B. Meyer and J. Woodcock, editors. *Verified Software: Tools, Theories, Experiments (proceedings of VSTTE conference, 2005)*, volume 4171 of *LNCS*. Springer-Verlag, 2007. To appear 2007.

[29] Microsoft. *Static Driver Verifier (SDV)*. `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`.

[30] H.H. Nguyen, C. David, S. Qin, and W.N. Chin. Automated verification of shape and size properties via separation logic. 2007. To appear at VMCAI'07.

[31] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[32] P. O'Hearn, J. Reynolds, and Y. Hongseok. Local reasoning about programs that alter data structures. In *Proceedings of CSL'01*, volume 2142 of *Lecture Notes in Computer Science*, Paris, 2001.

[33] S. Owre, N. Shankar, and J. Rushby. PVS: A prototype verification system. In D. Kapur, editor, *Proceedings of CADE-11*. Springer Verlag, 1992. LNAI vol. 607.

[34] V. Preoteasa. Mechanical verification of recursive procedures manipulating pointers using separation logic. TUCS Technical Report 753, Turku Centre for Computer Science, 2006.

[35] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[36] N. Schirmer. A Verification Environment for Sequential Imperative Programs in Isabelle/HOL. In G. Klein, editor, *Proc. NICTA Workshop on OS Verification 2004*, 2004.

[37] K. Slind, M. Gordon, R. Boulton, and A. Bundy. System description: An interface between CLAM and HOL. In C. Kirchner and H. Kirchner, editors, *15th International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 134–138, Lindau, Germany, July 1998. Springer. Earlier version available from Edinburgh as DAI Research Paper 885.

[38] H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, Nice, France, 2007.

[39] T. Weber. Towards mechanized program verification with separation logic. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic – 18th International Workshop, CSL 2004, 13th Annual Conference of the EACSL, Karpacz, Poland, September 2004, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 250–264. Springer, September 2004.

Lightweight Integration of the Ergo Theorem Prover inside a Proof Assistant *

Sylvain Conchon Evelyne Contejean
LRI, Univ Paris-Sud, CNRS
Orsay F-91405
INRIA Futurs, ProVal
Orsay, F-91893
{conchon,contejea}@lri.fr

Johannes Kanig Stéphane Lescuyer
INRIA Futurs, ProVal
Orsay, F-91893
LRI, Univ Paris-Sud, CNRS
Orsay F-91405
{kanig,lescuyer}@lri.fr

ABSTRACT

Ergo is a little engine of proof dedicated to program verification. It fully supports quantifiers and directly handles polymorphic sorts. Its core component is CC(X), a new combination scheme for the theory of uninterpreted symbols parameterized by a built-in theory X. In order to make a sound integration in a proof assistant possible, Ergo is capable of generating proof traces for CC(X). Alternatively, Ergo can also be called interactively as a simple oracle without further verification. It is currently used to prove correctness of C and Java programs as part of the Why platform.

1. INTRODUCTION

Critical software applications in a broad range of domains including transportation, telecommunication or electronic transactions are put on the market at an increasing rate. In order to guarantee the behavior of such programs, it is mandatory for a large part of the validation to be done in a mechanical way. In the ProVal project, we develop a platform [14] combining several tools of our own whose overall architecture is described in Figure 1. This toolkit enables the deductive verification of Java and C source code by generating *verification conditions* out of *annotations* in the source code. The annotations describe the logical specification of a program and the verification conditions are formulas whose validity ensures that the program meets its specification.

Much of the work of generating verification conditions for both Java and C programs is performed by Why, the tool which plays a central role in our toolkit and implements an approach designed by Filliâtre [13]. A main advantage of this architecture is that Why can output verification conditions to a large range of interactive higher-order provers (Coq, PVS, HOL, ...) and first-order automated provers such as CVC3 [3], Simplify[11], Yices [10], Z3 [9] or Ergo [4].

When using first-order automatic provers, a great number of formulas can be discharged in very little time in comparison to the tedious process of interactively proving these formulas in Coq or Isabelle.

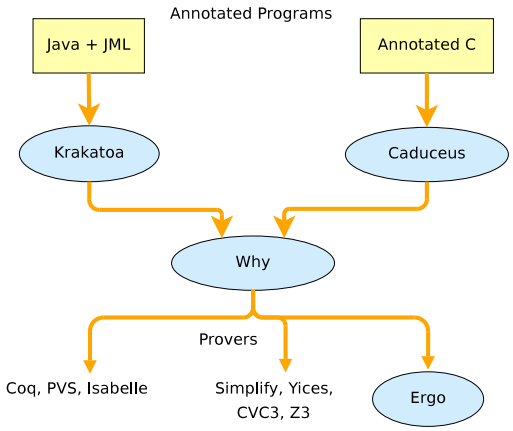


Figure 1: The Proval tool chain

The immediate downside of this method is that the soundness then depends on the soundness of the automated provers, which weakens the chain of trust. Depending on whether the automated provers shall be trusted or not, there are different ways of integrating them in the system : some may want to use automated provers as "black boxes" and invoke them from within the prover (like PVS does), while others will rather have the prover produce some *traces* of its proofs and typecheck these traces. The problem with the latter is that the production of a complete proof term (as done by the Omega or Zenon tactics in Coq) can be a slow and difficult process. Thus, the production of small, efficient traces is the cornerstone of the sound integration of an automated prover in an interactive prover. Another, more practical, issue raised by using automated theorem provers is that verification conditions generated by Why are expressed in a polymorphic first-order logic, while existing provers only handle untyped logic (such as Simplify and HaRVey) or monomorphic many-sorted logic (such as Yices, CVC3). It has been shown in [8] that finding encodings between these logics which are correct and do not deteriorate the performance of the provers is not a trivial issue.

Therefore, we have developed the Ergo theorem prover with these different limitations in mind; the main novelties in our system are the native support of polymorphism, a new modular congruence

* Work partially supported by A3PAT project of the French ANR (ANR-05-BLAN-0146-01).

AFM'07, November 6, Atlanta, GA, USA.

closure algorithm CC(X) for combining the theory of equality over uninterpreted symbols with a theory X, and a mechanism producing lightweight proof traces.

The remainder of this paper focuses on the design of Ergo and attempts of integration in Coq. Section 2 describes the different characteristics and central components in Ergo, whereas Section 3 details both a loose integration of Ergo in Coq, and a tighter integration based on the production of efficient traces for the *congruence closure* module of Ergo.

2. THE ERGO THEOREM PROVER

Ergo is an automatic theorem prover fully integrated in the program verification tool chain developed in our team. It solves goals that are directly written in the Why's annotation language [1]. This means that Ergo fully supports quantifiers and deals directly with polymorphism.

2.1 General Architecture

The architecture of Ergo is highly modular: each part (except the parsers) of the code is described by a small set of inference rules and is implemented as a (possibly parameterized) module. Figure 2 describes the dependencies between the modules. Each input syntax is handled by the corresponding parser. Both of them produce an abstract syntax tree in the same datatype. Hence, there is a single typing module for both input syntaxes. The main loop consists of three modules:

- A home-made efficient SAT-solver with backjumping that also keeps track of the lemmas of the input problem and those that are generated during the execution.
- A module that handles the ground and monomorphic literals assumed by the SAT-solver. It is based on a new combination scheme,CC(X), for the theory of uninterpreted symbols and built-in theories such as linear arithmetic, the theory of lists etc.
- A matching module that builds monomorphic instances of the (possibly polymorphic) lemmas contained in the SAT-solver modulo the equivalence classes generated from the decision procedures.

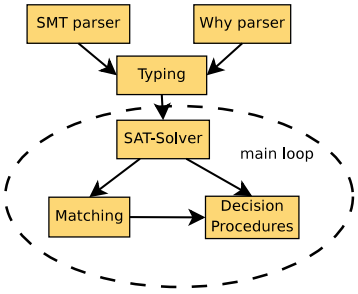


Figure 2: The modular architecture of Ergo.

The rest of this section explains the core decision procedures and how quantifiers are supported by Ergo[2] and their subtle interaction with polymorphism.

[1]Ergo also parses the standard [17] defined by the SMT-lib initiative.

[2]Ergo handles quantifiers in a very similar way to Simplify and Yices.

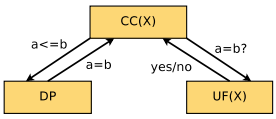


Figure 3: CC(X) architecture

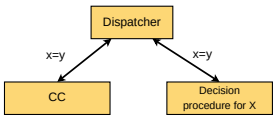


Figure 4: Nelson-Oppen architecture

2.2 Built-In Decision Procedures

The decision procedure module implements CC(X) [5], a new combination scheme *à la Shostak* [19, 18] between the theory of uninterpreted symbols and a theory X. CC(X) means "congruence closure parameterized by X". The module X should provide a decision procedure DP for its relational symbols except for the equality which is handled by a generic union-find algorithm, UF(X), parameterized by X. As shown in Figure 3, the combination relies on the following exchanges:

- CC(X) sends relations between *representatives* in UF(X) to DP. Using representatives automatically propagates the equalities implied by UF(X). In return, DP sends its discovered equalities.
- CC(X) *asks* UF(X) *for relevant equalities* to propagate for congruence. Due to the union-find mechanism, asking for relevant equalities is much more efficient than letting UF(X) try to discover *all* new equalities.

This is different from the Nelson-Oppen combination [15] where, as shown in Figure 4, the combined modules have to discover and propagate all their new equalites.

Currently, CC(X) has been instantiated by linear arithmetic [3], the theory of lists, the general theory of constructors and a restricted theory of accessibility in graphs [7].

2.3 Quantifiers

The SAT-solver module takes as input CNF formulas, seen as sets of disjunctions where leaves are either ground literals or quantified formulas in prenex normal form.

A (standard) propositional SAT engine decides the satisfiability of a propositional formula by assuming (positively or negatively) each leaf of the associated CNF, and then by simplifying the formula accordingly to these choices. It stops whenever the CNF becomes equal to the empty set, proving that the input formula is satisfiable by providing a model. In Ergo, the general mechanism of the SAT-solver is quite similar, but there are two main differences.

The way the leaves are handled depends on their nature: assuming a ground literal amounts to sending it to the decision procedure module, while assuming positively a quantified formula simply means to store it in the current state of the SAT engine.

When the CNF is empty, the Ergo SAT-solver still has to handle the previously stored quantified formulas. In general, it is obviously not possible to decide whether these formulas are consistent

[3]Ergo is complete over rationals but uses heuristics for integers.

with the partial model already built. However, one may try to prove inconsistency by using ground instances of these formulas. The instantiation mechanism is provided by the matching module which builds a new CNF by instantiating the quantified formulas with some ground terms *occurring in the ground literals already handled*. A pattern mechanism (similar to Simplify's triggers) is used to guide quantifier instantiation. Patterns can either be defined by the user or automatically generated.

2.4 Polymorphism à la ML

In Ergo, the matching module also handles the polymorphism by instantiating type variables. Consider for instance the following example written in the Why syntax which defines the sort of polymorphic lists (α `list`) and its constructors (`nil` and `cons`) as well as a function `length` with its properties (`a1` and `a2`).

```
type α list
logic nil: α list
logic cons: α, α list → α list
logic length: α list → int

axiom a1: length(nil) = 0
axiom a2: ∀x:α. ∀l:α list.
            length(cons(x,l)) = 1 + length(l)
```

First, the typing module checks that this input is well-typed, and when encountering a goal such as

```
goal g: ∀x:α. length(cons(x,nil))=1
```

it turns the term variable *x* into a constant *a* (usual transformation) as well as the implicitly universally quantified type variable α into a type constant τ. This implies in particular that the context contains the type constant τ and the term constant *a* of type τ and that the `goal g':length(cons(a,nil))=1` is monomorphic.

Now, in order to prove the goal `g'`, the matching module generates the instance of `a2` by the substitution

```
{α → τ, x → a, l → nil}
```

We are left to prove that `1+length(nil)=1` where `nil` has the type τ `list`. The only way to show this is by using `a1`.

At first glance, `a1` seems to be a monomorphic ground literal that could be sent to the decision procedure module and not a lemma, since no explicit quantified variable occurs in it. However if it is considered as such, the type of `nil` is fixed to an arbitrary constant which is *distinct* from τ. This prevents using `a1` to conclude that `1+length(nil)=1` holds when `nil` has type τ `list`. The actual lemma should be:

```
axiom a1' : ∀α. length(nil:α list) = 0
```

but in the Why syntax, the type variables such as α are only implicitly universally quantified. Some of these type variables occur explicitly in the types of the quantified term variables (such as `x:`α in `a2`), but others are hidden in polymorphic constants. This is the case for the type variable α of the constant `nil` in the axiom `a1`, which has to be internally translated by Ergo into `a1'`.

To sum up, there is an invariant in the main loop:

1. a goal is always monomorphic;
2. only monomorphic ground literals are sent by the SAT-solver to the decision procedures' module;
3. the matching module instantiates polymorphic lemmas using the monomorphic ground types and terms already handled, thus the generated instances are monomorphic.

2.5 General Benchmarks

Ergo is written in Ocaml and is very light ($\sim$ 3000 lines of code). It is freely distributed under the Cecill-C licence at `http://ergo.lri.fr`.

Ergo's efficiency mostly relies on the technique of hash-consing. Beyond the obvious advantage of saving memory blocks by sharing values that are structurally equal, hash-consing may also be used to speed up fundamental operations and data structures by several orders of magnitude when sharing is maximal. The hash-consing technique is also used to elegantly avoid the blow-up in size due to the CNF conversion in the SAT-solver [6].

Since the built-in decision procedures are tightly coupled to the top-level SAT-solver, the backtracking mechanism performed by the SAT module forces the decision procedure module to come back to its previous state. This is efficiently achieved by using functional data structures of Ocaml.

We benchmarked Ergo on a set of 1450 verification conditions that were automatically generated by the VCG Caduceus/Why from 69 C programs [16]. These goals make heavy use of quantifiers, polymorphic symbols and linear arithmetic. All these conditions are proved at least by one prover. Figure 5 shows the results of the comparison between Ergo and four other provers: Z3, Yices, Simplify and CVC3. As mentioned above, none of these provers can directly handle polymorhism; therefore we simply erased types for Simplify and we used an encoding for Yices, Z3 and CVC3. The five provers were run with a fixed timeout of 20s on a machine with Xeon processors (2.13 GHz) and 2 Gb of memory.

	valid	**timeout**	**unknown**	**avg. time**
Simplify v1.5.4	98%	1%	1%	60ms
Yices v1.0	95%	2%	3%	210ms
Ergo v0.7	94%	5%	1%	150ms
Z3 v0.1	87%	10%	3%	690ms
CVC3 v20070307	71%	1%	28%	80ms

Figure 5: Comparison between Ergo, Simplify, Yices and CVC-Lite on 1450 verification conditions.

The column **valid** shows the percentage of the conditions proved valid by the provers. The column **timeout** gives the percentage of timeouts whereas **unknown** shows the amount of problems unsolved due to incompleteness. Finally, the column **avg. time** gives the average time for giving a valid answer.

As shown by the results in Figure 5, the current experimentations

are very promising with respect to speed and to the number of goals automatically solved.

3. INTEGRATION OF ERGO IN COQ

Today, Coq still lacks good support of proof automation. There are two main reasons for that. On the one hand, Coq's rich higher-order logics is not well-adapted to decision procedures that were designed for first order logics. On the other hand, Coq is built following the de Bruijn principle: any proof is checked by a small and trusted part of the system. Making a complex decision procedure part of the trusted system would go against this principle.

Still, one would like to use automated provers such as Ergo in Coq. We briefly present two possible approaches to this problem, one by giving up the de Bruijn principle, the other one by maintaining it.

3.1 A loose integration

To be able to use Ergo in Coq, one has to translate goals in Coq higher order logic into first order logic, understood by Ergo. Ayache and Filliâtre have realized such a translation [2]. In their approach, the Coq goal is translated, sent to Ergo, and the answer of the automated prover is simply trusted. Here, by using the automated prover as an *oracle*, the de Bruijn principle is given up, but the resulting Coq tactics are quite fast.

This translation aims the polymorphic first order logic of the Why tool, which is the same logic as the one of Ergo. It not only translates terms and predicates of the Coq logic CIC, the Calculus of Inductive Constructions, but also includes several techniques to go beyond: abstractions of higher-order subterms, case analysis, mutually recursive functions and inductive types.

Using the Why syntax as target language has the advantage of being able to interface any automated prover supported by Why with the above translation. Simplify, Yices, CVC Lite and other provers may be called from Coq. The first order prover Zenon even returns a proof trace in form of a Coq term. The soundness of its answers can thus be checked by Coq.

3.2 A tight integration via traces

The oracle approach above has the disadvantage that it is very easy to introduce bugs in the translation or in the prover, which may compromise a whole proof development in Coq. Constructing a Coq proof term directly is sound, but may generate huge proof traces and is difficult if the problems contain interpreted function symbols of some theory, for example the theory of linear arithmetic (the tool Zenon mentioned above does not handle arithmetic). Another approach consists in modelling part of the prover in Coq and only communicating applications of inference rules or other facts that are relevant for soundness. In particular, any part of the execution that is concerned with proof search can be omitted. The size of the proof is expected to be considerably shorter, and thus the time to check this proof. Proofs that are guided by the execution of the decision procedure are called *traces*. This section describes the ongoing work of constructing proof traces for the core decision procedure $\mathsf{CC}(\mathsf{X})$.

As described in section 2.2, the core decision procedure $\mathsf{CC}(\mathsf{X})$ of Ergo uses a module $\mathsf{UF}(\mathsf{X})$ to handle the equality axioms, ie. reflexivity, symmetry and transitivity, as well as equality modulo the theory X. If the soundness of such a union-find module is established, and we are given a set E of initial equations, a sound concrete union-find structure can be constructed as follows: starting with the empty union-find structure (that only realizes equality modulo X), we only have the right to process (merge) equations that are either in E or are of the form $f(a_1,\cdots,a_n) = f(b_1,\cdots,b_n)$, where the representatives of a_i and b_i are the same, for all i. This translates directly into an inductive type definition in Coq:

```
Inductive Conf (e:list equation) : uf → Set :=
 | init : Conf e Uf.empty
 | in_p : ∀(u:uf) (t1 t2:term),
           Conf e u → In (t1,t2) e →
             Conf e (Uf.union u t1 t2)
 | congr : ∀(u:uf) (l1 l2: list term) (f: symbol),
       Conf e u → list_eq (Uf.equal u) l1 l2 →
       Conf e (Uf.union u (Term f l1) (Term f l2)).
```

The function `list_eq` takes a relation as first argument and returns the relation lifted to lists; otherwise this definition should be self explanatory.

Now, for any object of type `Conf e u`, it is not difficult to prove in Coq that the union-find structure `u` is indeed sound, by proving the following lemma:

```
Theorem correct_cc :
  ∀(u:uf) (e :list equation),
    Conf e u →
     (∀(t1 t2:term), Uf.equal u t1 t2 →
       Th.thEX e t1 t2).
```

where `X.thEX` is the target relation $=_{E,X}$, which means equality modulo the theory X and the set E of assumed ground equations. The Coq proof of theorem `correct_cc` follows the paper proof of soundness of $\mathsf{CC}(\mathsf{X})$ given in the appendix of [5] and consists of 150 lines of specification and 300 lines of proof script.

This enables us to construct a proof from a run of Ergo: by recording the processing of equations in the $\mathsf{CC}(\mathsf{X})$ module, establish a Coq object of type `Conf e u`, deliver a proof that this union-find module renders t_1 equal to t_2, and by the application of the theorem `correct_cc` we obtain a proof for $t_1 =_{E,X} t_2$. If the union-find module is actually implemented in Coq, we can even obtain the proof of t_1 and t_2 having the same representative automatically, by a technique called *reflection* that employs the calculating capabilities of the proof assistant.

Thus, all that is left is an implementation of a union-find modulo a theory X in Coq (UF_{Coq} in the following), which in turn requires the implementation of the theory X_{Coq}. To be independent of any particular theory, we use the same trick as $\mathsf{CC}(\mathsf{X})$ uses: we develop a parameterized module (a *functor*), that may be instantiated by *any* theory that provides the necessary constructs. With the strong type system of Coq, we can even express and require soundness properties of the theory that are necessary to prove the soundness of $\mathsf{UF}_{Coq}(\mathsf{X}_{Coq})$.

A subtlety in the implementation of proof traces is the handling of function symbols and constants. On the one hand, one would like to be as flexible as possible and not fix the set of used function symbols in advance (in general, every problem will use its own set of symbols). On the other hand, it is necessary to reason about

some (fixed) function symbols, like $+$ in the case of arithmetic. Our solution to this dilemma is (again) the use of a functor: the theory X_{Coq} is parameterized by a signature S which provides uninterpreted function symbols. Internally, the theory completes this signature with its own function symbols and may then reason about the resulting signature. To obtain the union-find structure in Coq, we can now instantiate UF_{Coq} by $\mathsf{X}_{Coq}(S)$. In practice, the signature S is generated automatically by the proof traces generation mechanism. To summarize, we obtain the following instantiation chain:

$$S \rightarrow \mathsf{X}_{Coq}(S) \rightarrow \mathsf{UF}_{Coq}(\mathsf{X}_{Coq}(S)) \rightarrow \mathsf{CC}_{Coq}$$

To avoid reconstructing CC_{Coq} for identical signatures and theories, the instantiation may be part of a prelude file, as is already the case for type definitions, axioms, etc. in the VCGs generated by the Why tool.

4. CONCLUSION

We have presented Ergo, a new theorem prover for first-order polymorphic logic with built-in theories. The development started in January 2006 and the current experimentations are very promising with respect to speed and to the number of goals automatically solved.

We also described two attempts at integrating Ergo in Coq: first as an oracle, which raises concerns about the soundness of the certification chain, and then we showed how to generate proof traces for the congruence closure algorithm of Ergo so as to let Coq verify the proof itself. Such traces are a very interesting way of mechanizing interactive proving without breaking the chain of trust [1]. Our first experiments with the traces generation are promising: the generic nature of $\mathsf{CC}(\mathsf{X})$ is truly captured and traces are concise. There is room for improvement in terms of efficiency and traces should ideally also cover the SAT-solver and matching components. For the moment, only traces for linear arithmetic are implemented. In total, the Coq development of the theory of linear arithmetic takes about 400 lines of specification and 900 lines of proof script.

Another direction, that we think is worth investigating, is to "prove the prover" in a proof assistant. Indeed, Ergo uses only purely functional data-structures (with the exception of the hash-consing modules), is highly modular and very concise ($\sim$ 3000 lines of code). All these features should make a formal certification feasible.

Also, since this prover is partly dedicated to the resolution of verification conditions generated by the Krakatoa/Caduceus/Why[14] toolkit, its future evolution is partly guided by the needs of these tools : designing efficient proof strategies to manage huge contexts and useless hypotheses and adding more built-in theories such as pointer arithmetic. We also plan to design parsers in order to run Ergo on other benchmarks such as ESC/Java, Boogie and NASA benchmarks.

Finally, we are currently working on a functor `Combine(X1,X2)` to effectively combine different built-in theories `X1` and `X2` under certain restrictions, and by taking advantage of the fact that our theories are typed.

5. REFERENCES

[1] The A3PAT Project. `http://www3.iie.cnam.fr/~urbain/a3pat/`.

[2] N. Ayache and J.-C. Filliâtre. Combining the Coq Proof Assistant with First-Order Decision Procedures. Unpublished, March 2006.

[3] C. Barrett and C. Tinelli. Cvc3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany*, Lecture Notes in Computer Science. Springer, 2007.

[4] S. Conchon and E. Contejean. The Ergo automatic theorem prover. `http://ergo.lri.fr/`.

[5] S. Conchon, E. Contejean, and J. Kanig. CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In S. Krstic and A. Oliveras, editors, *SMT 2007: 5th International Workshop on Satisfiability Modulo*, 2007.

[6] S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *ACM SIGPLAN Workshop on ML*, Portland, Oregon, Sept. 2006.

[7] S. Conchon and J.-C. Filliâtre. Semi-Persistent Data Structures. Research Report 1474, LRI, Université Paris Sud, September 2007.

[8] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In *21th International Conference on Automated Deduction (CADE-21)*, volume 4603 of *LNCS (LNAI)*, pages 263–278, Bremen, Germany, July 2007.

[9] L. de Moura and N. Bjørner. Z3, An Efficient SMT Solver. `http://research.microsoft.com/projects/z3/`.

[10] L. de Moura and B. Dutertre. Yices: An SMT Solver. `http://yices.csl.sri.com/`.

[11] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[12] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. PhD thesis, Université Paris-Sud, July 1999.

[13] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. 13(4):709–745, July 2003. [English translation of [12]].

[14] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification.

[15] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming, Languages and Systems*, 1(2):245–257, Oct. 1979.

[16] ProVal Project. Why Benchmarks. `http://proval.lri.fr/why-benchmarks/`.

[17] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `http://www.SMT-LIB.org`, 2006.

[18] H. Rueß and N. Shankar. Deconstructing Shostak. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 19, Washington, DC, USA, 2001. IEEE Computer Society.

[19] R. E. Shostak. Deciding combinations of theories. *J. ACM*, 31:1–12, 1984.

Using SMT solvers to verify high-integrity programs*

Paul B. Jackson
School of Informatics
University of Edinburgh
King's Bldgs
Edinburgh, EH9 3JZ
United Kingdom
Paul.Jackson@ed.ac.uk

Bill J. Ellis
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton
Edinburgh, EH14 4AS
United Kingdom
bill@macs.hw.ac.uk

Kathleen Sharp
IBM United Kingdom Ltd
Hursley House, Hursley Park
Winchester, SO21 2JN
United Kingdom
Kathleen.sharp@uk.ibm.com

ABSTRACT

In this paper we report on our experiments in using the currently popular SMT (SAT Modulo Theories) solvers Yices [10] and CVC3 [1] and the Simplify theorem prover [9] to discharge verification conditions (VCs) from programs written in the SPARK language [5]. SPARK is a subset of Ada used primarily in high-integrity systems in the aerospace, defence, rail and security industries. Formal verification of SPARK programs is supported by tools produced by the UK company Praxis High Integrity Systems. These tools include a VC generator and an automatic prover for VCs.

We find that Praxis's prover can prove more VCs than Yices, CVC3 or Simplify because it can handle some relatively simple non-linear problems, though, by adding some axioms about division and modulo operators to Yices, CVC3 and Simplify, we can narrow the gap. One advantage of Yices, CVC3 and Simplify is their ability to produce counter-example witnesses to VCs that are not valid.

This work is the first step in a project to increase the fraction of VCs from current SPARK programs that can be proved automatically and to broaden the range of properties that can be automatically checked. For example, we are interested in improving support for non-linear arithmetic and automatic loop invariant generation.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*mechanical verification, assertions, invariants, pre- and post-conditions*

General Terms

Experimentation, Performance, Verification

*This work was funded in part by UK EPSRC Grant GR/S01771/01.

AFM '07, November 6, Atlanta, GA, USA.

Keywords

SMT solver, SAT modulo theories solver, Ada, SPARK

1. INTRODUCTION

SMT (SAT Modulo Theories) solvers combine recent advances in techniques for solving propositional satisfiability (SAT) problems [33] with the ability to handle first-order theories using approaches derived from Nelson and Oppen's work on cooperating decision procedures [24]. The core solvers work on quantifier free problems, but many also can instantiate quantifiers using heuristics developed for the non-SAT-based prover Simplify [9]. Common theories that SMT solvers handle include linear arithmetic over the integers and rationals, equality, uninterpreted functions, and datatypes such as arrays, bitvectors and records. Such theories are common in VCs and so SMT solvers are well suited to automatically proving VCs.

SMT solvers are currently used to prove Java VCs in the ESC/Java2 [2] tool and C# VCs in the Spec# static program verifier [6]. The Simplify prover was used to prove VCs in the ESC/Modula-3 and original ESC/Java tools.

We evaluate here the current releases of two popular SMT solvers: CVC3 [1] and Yices [10]. Both these systems featured in the 2006 and 2007 SMT-COMP competitions comparing SMT solvers[1] in the category which included handling quantifier instantiation. We also evaluate Simplify because it is highly regarded and, despite its age (the latest public release is over 5 years old), it is still competitive with current SMT solvers.

One advantage that SMT solvers and Simplify have over Praxis's Simplifier is their ability to produce counter-example witnesses to VCs that are not valid. These counter-examples can be of great help to SPARK program developers and verifiers: they can point out scenarios highlighting program bugs or indicate what extra assertions such as loop invariants need to be provided. They also can reduce wasted time spent in attempting to interactively prove false VCs.

The work reported here is the first step in a project to improve the level of automation of SPARK VC verification and extend the range of properties that can be automatically verified. Typical properties that SPARK users currently verify are those which check for the absence of run-time exceptions caused by arithmetic overflow, divide by zero, or array bounds violations. Particular new kinds of properties we are interested in include those which involve non-linear

[1] `http://www.smtcomp.org/`

arithmetic and which involve significant quantifier reasoning. A key obstacle to improving automation is the need to provide suitable loop invariants. We are therefore interested in also exploring appropriate techniques for automatic invariant generation.

Tackling SPARK programs rather than say Java or C programs is appealing for a couple of reasons. Firstly, there is a community of SPARK users who have a need for strong assurances of program correctness and who are already writing formal specifications and using formal analysis tools. This community is a receptive audience for our work and we have already received strong encouragement from Praxis. Secondly, SPARK is semantically relatively simple and well defined.

Notable earlier work on improving the verification of SPARK programs is by Ellis and Ireland [18]. They used a proof-planning and recursion analysis approach to analyse failed proofs of VCs involving loops to identify how to strengthen loop invariants.

Some success has been had at NASA in using first-order automatic theorem provers for discharging VCs [8]. A major problem with such an approach is the poor support for arithmetic in such provers. This work succeeds by axiomatically characterising a fragment of linear arithmetic that is just sufficient for the kinds of VCs encountered.

Section 2 gives more background on SPARK. Section 3 describes the current implementation of our interface to Yices and CVC3. Case study programs are summarised in Section 4 and Sections 5 and 6 present the results of experiments on the VCs from these programs. Future work, both near term and longer term is discussed in Section 7 and conclusions are in Section 8.

2. THE SPARK LANGUAGE AND TOOLKIT

The SPARK [5] subset of Ada was first defined in 1988 by Carré and Jennings at Southampton University and is currently supported by Praxis. The Ada subset was chosen to simplify verification: it excludes features such as dynamic heap-based data-structures that are hard to reason about automatically. SPARK adds in a language of program annotations for specifying intended properties such as pre and post conditions and assertions. These annotations take the form of Ada comments so SPARK programs are compilable by standard Ada compilers.

A feature of SPARK inherited from Ada particularly relevant for verification purposes is the ability to specify subtypes, types which are subranges of integer, floating-point and enumeration types. For example, one can write:

```
subtype Index is Integer range 1 .. 10;
```

Such types allow programmers to easily include in their programs extra specification constraints without having to supply explicit annotations. It is then possible, either dynamically or statically, to check that these constraints are satisfied.

The Examiner tool from Praxis generates verification conditions from programs. Annotations are often very tedious for programmers to write, so the Examiner can generate automatically some kinds of annotations. For example, it can generate annotations for checking the absence of run-time errors such as array index out of bounds, arithmetic overflow, violation of subtype constraints and division by zero.

As with Ada, a SPARK program is divided into program units, each usually corresponding to a single function or procedure. The Examiner reads in files for the annotated source code of a set of related units and writes the VCs for each unit into 3 files:

- A *declarations* file declaring functions and constants and defining array, record and enumeration types,
- a *rules* file assigning values to constants and defining properties of datatypes,
- a *verification condition goal* file containing a list of verification goals. A goal consists of a list of hypotheses and one or more conclusions. Conclusions are implicitly conjuncted rather than disjuncted as in some sequent calculi.

The language used in these files is known as FDL.

The Simplifier tool from Praxis can automatically prove many verification goals. It is called the *Simplifier* because it returns simplified goals in cases when it cannot fully prove the goals generated by the Examiner. Users can then resort to an interactive proof tool to try to prove these remaining simplified goals. In practice, this proof tool requires rather specialised skills and is used much less frequently than the Simplifier. To avoid the need to use the tool or to manually review verification goals, users are often inclined to limit the kinds of program properties they try to verify to ones that can be verified by the Simplifier. They also learn programming idioms that lead to automatically provable goals.

3. SMT SOLVER INTERFACE

Our interface program reads in the VC file triples output by the Examiner tool and generates a series of *goal slices*. A goal slice is a single conclusion packaged with its associated hypotheses, rules and declarations. After suitable processing, each goal slice is passed in turn to a selected prover, at present one of Yices, CVC3 or Simplify. The processing includes:

- Handling enumerated types.

  The Examiner generates rules that define each enumerated type as isomorphic to a subrange of the integers. Explicit functions defining the isomorphism are declared and the rules for example relate order relations and successor functions on the enumeration types to the corresponding relations and functions on the integers.

  We experiment with 3 options:

  1. Mapping each FDL enumerated type to the enumerated type of the SMT solver. We keep the rules since neither solver publically supports ordering of the elements of enumerated types.
  2. Mapping each FDL enumerated type to an abstract type in the SMT solver, so the solver has to rely fully on the supplied enumeration type rules to reason about the enumeration constants.
  3. Defining each enumeration type as an integer subrange, defining each type element as equal to an appropriate integer, and eliminating all rules.

The last option generally gives the best performance, but the first two are better from a counter-example point of view: counter-examples involving the enumeration types are more readable since they use the enumeration constants rather than the corresponding integers.

- Resolving overloaded functions and relations.

  For example, FDL uses the same symbols for the order relations and successor functions on all types. Resolution involves inferring types of sub-expressions and type checking the FDL.

- Inferring types of free variables in rules and adding quantifiers to close the rules.

  FDL is rather lax in declaring types of these variables and so types must be inferred from context.

- Resolving the distinction between booleans as individuals and booleans as propositions. The FDL language uses the same type for both, as does Yices. However, CVC3 and Simplify take a stricter first-order point of view and require the distinction.

- Adding missing declarations of constants. FDL has some built-in assumptions about the definition of constants for the lowest and highest values in integer and real subrange types and we needed to make these explicit.

- Reordering type declarations so types are defined before they are used. Such an ordering is not required in FDL, but is needed by Yices and CVC3.

We carry out all the above processing in a prover independent setting as much as possible in order to keep the driver code for individual provers compact and ease the development of further drivers.

The match between the FDL language of the SPARK VCs and the input languages of both Yices and CVC3 is good. The FDL language includes quantified first-order formulae and datatypes including the booleans, integers, reals, enumerations, records and arrays, all of which are supported by both solvers.

At present we are interfacing to Yices and CVC3 using their C and C++ APIs respectively. An alternative is to write goal slices to files in the specific input languages of the respective solvers and call the solvers in sub-processes. We take this latter approach with Simplify since it does not have a readily-available C or C++ API.

Simplify's input language is rather different from FDL. All individual expressions in Simplify are untyped except for those which are arguments to arithmetic relations or are the arguments or results of arithmetic operations - these expressions are of integer type. We handle both arrays and records using Simplify's built-in axiomatic theory of maps. For example, one axiom is stated as:

```
(FORALL (m i x)
  (PATS (select (store m i x) i))
  (EQ (select (store m i x) i) x)
)
```

Here the `PATS` expression is a hint used to suggest to Simplify a sub-expression to use as a trigger pattern when searching for matches that could provide instantiations. Handling enumeration types is straightforward: the FDL generated by Praxis's Examiner provides enough inequality predicates bounding enumeration type values to allow the interpretation of enumeration types themselves as integers. A problem with Simplify is that all arithmetic is fixed precision. We follow the approach taken when Simplify is used with ESC/Java where all constants with magnitude greater than some threshold are represented symbolically and axioms are asserted concerning how these constants are ordered [21].

To aid in analysis of results, we provide various options for writing information to a log file, as well as writing comma-separated-value run summaries. These allow easy comparison between results from runs with different options and solvers.

4. CASE STUDY SPARK PROGRAMS

For our experiments we work with two readily available examples.

- **Autopilot**: the largest case study distributed with the SPARK book [5]. It is for an autopilot control system for controlling the altitude and heading of an aircraft.

- **Simulator**: a missile guidance system simulator written by Adrian Hilton as part of his PhD project. It is freely available on the web[2] under the GNU General Public Licence.

Some brief statistics on each of these examples and the corresponding verification conditions are given in Table 1. The

	Autopilot	Simulator
Lines of code	1075	19259
No. units	17	330
No. annotations	17	37
No. VC goals	133	1799
No. VC goal slices	576	6595

Table 1: Statistics on Case Studies

lines-of-code estimates are rather generous, being simply the sum of the number of lines in the Ada specification and body files for each example. The *annotations* count is the number of SPARK precondition and assertion annotations in all the Ada specification and body files. No postconditions were specified in either example.

In both cases the VCs are primarily from exception freedom checks, e.g. checking that arithmetic values and array indices are always appropriately bounded.

The VCs from both examples involve enumerated types, linear and non-linear integer arithmetic, integer division and uninterpreted functions. In addition, the Simulator example includes VCs with records, arrays and the modulo operator.

5. EXPERIMENTAL RESULTS

Results are presented for the tools

- Yices 1.0.9,

- CVC3 development version 20071001,

- Simplify 1.5.4,

[2] `http://www.suslik.org/Simulator/index.html`

	Yices		CVC3		Simplify		Simplifier	
proven	510	88.5%	518	89.9%	516	89.6%	572	99.3%
unproven	66	11.5%	58	10.1%	60	10.4%	4	0.7%
timeout	0	0 %	0	0 %	0	0 %	0	0 %
error	0	0 %	0	0 %	0	0 %	0	0 %
total	576		576		576		576	

Table 2: Coverage of Autopilot goal slices

	Yices		CVC3		Simplify		Simplifier	
proven	6004	91.0%	6074	92.1%	5940	90.1%	6410	97.2%
unproven	591	9.0%	337	5.1%	646	9.8%	185	2.8%
timeout	0	0 %	184	2.8%	0	0 %	0	0 %
error	0	0 %	0	0 %	9	0.1%	0	0 %
total	6595		6595		6595		6595	

Table 3: Coverage of Simulator goal slices

- Simplifier 2.32, part of the 7.5 release of Praxis's tools.

We needed a development version of CVC3 as the latest available release (1.2.1) had problems with a significant fraction of our VCs.

Unless otherwise stated, the experiments were run on a 3GHz Pentium 4D CPU with 1GB of physical memory running Linux Fedora Core 6. We used the translation option to eliminate the enumeration type occurrences, as this yields the best performance. With Simplify, the threshold for making constants symbolic was set at 100,000.

The coverage obtained with each tool is summarised in Tables 2 and 3. The tables show the results of running Yices, CVC3, Simplify and Praxis's Simplifier on each goal slice from the VCs in each of the case studies. The *proven* counts are for when the prover returned claiming that a goal is true. The *unproven* counts are for when the prover returned without having proven the goal. The *timeout* counts are for when the prover reached a time or resource limit. The only limit reached in the tests was a resource limit for CVC3: CVC3 usually reached the set limit of 100,000 in 8-10 sec. This limit was only specified for the CVC3 runs on the Simulator VCs. The *error* counts are for when the prover had a segmentation fault, encountered an assertion failure or diverged, never reaching a resource limit after a few minutes.

To indicate the performance of each prover, we have gathered and sorted run times of each prover on each goal slice. Table 4 shows the run times at a few percentiles. 'TO' indicates that the timeout resource limit was reached. With the CVC3 Simulator runs, this was reached at the 97% level.

	Autopilot			Simulator		
	Yices	CVC3	Smpfy	Yices	CVC3	Smpfy
50%	.01	.02	.01	.01	.05	.04
90%	.02	.04	.02	.03	.11	.07
99%	.02	4.75	.03	.04	TO	.08
99.9%				.16		.08
max.	.03	17.20	.03	.56	>10	.09

Table 4: Run time distributions (times in sec.)

Numbers are not given for the Simplifier in this table as it does not provide a breakdown of its run time on individual goal slices.

Table 5 provides a comparison of the total run times of each prover on all goal slices. These times also include the overhead time for reading in the various VC files and suitably translating them. For Yices and CVC3, this overhead is at most a few percent.

	Autopilot	Simulator
Yices	5.63	109.6
CVC3	83.11	2928.0
Simplify	8.09	293.1
Simplifier	6.51	226.9

Table 5: Total run times (sec)

The number for Praxis's Simplifier running on the Simulator goal slices is an estimate based on running a Solaris version of the Simplifier on a slower SPARC machine and obtaining a scaling factor by running both Solaris and Linux versions on the Autopilot goal slices. Praxis only release a Linux version of the Simplifier for demonstration purposes, and this version cannot handle the size of the Simulator example.

Both the total and individual goal slice times for Simplify appear to be significant over-estimates of the time spent executing Simplify's code. For example, a preliminary investigation shows that for only 15-30% of the total run times for the Simplify tests is the CPU in user mode executing the child processes running Simplify. Much of the rest of the time seems to be spent in file input/output (communication with Simplify is via temporary files) and sub-process creation and clean-up.

6. DISCUSSION OF RESULTS

Coverage.

All the goal slices unproven by Yices, CVC3 or Simplify, but certified true by Praxis's Simplifier, involve non-linear arithmetic with some combination of non-linear multiplication, integer division and modulo operators. These slices nearly all involve checking properties to do with bounds on the values of expressions. It is fairly simple to see these properties are true from considering elementary bounding properties of arithmetic functions, for example, that the product of two non-negative values is non-negative. See the section below on *incompleteness* for a discussion of an experiment

involving adding axioms concerning bounding properties.

The 4 Autopilot goal slices unproven by Simplifier are all true. They all have similar form: for example, one goal slice in essence in FDL syntax is:

```
H1:    f > 0 .
H2:    f <= 100 .
H3:    v >= 0 .
H4:    v <= 100 .
       ->
C1:    (100 * f) div (f + v) <= 100 .
```

The 2.8% Simulator goal slices unproven by Simplifier appear to be nearly all false. They are derived from 47 of the 330 Simplifier sub-programs. The author of the Simulator case study code had neither the time nor the need to ensure that all goal slices for all sub-programs were true.

The slightly better coverage obtained with CVC3 over Yices on the Autopilot example is partly because Yices prunes any hypothesis or conclusion with a non-linear expression, whereas CVC3 accepts non-linear multiplication and knows some properties of it. For example, it proved the goal slices:

```
H1:    s >= 0 .
H2:    s <= 971 .
        ->
C1:    43 + s * (37 + s * (19 + s)) >= 0 .
C2:    43 + s * (37 + s * (19 + s)) <= 214783647 .
```

Run times.

As can be seen from the distributions, Yices, CVC3 and Simplify all have run times within a factor of 5 of each other on many problems. CVC3's total run times at 15-20× those of Yices seems to be due to it trying for longer on problems where it returns *unproven* or *timeout* results: all but 16 of the problems it proves are proven in under 0.11s.

The performance of Simplify is impressive, especially given its age (the version used dates from 2002) and that it does not employ the core SAT algorithms used in the SMT solvers. Part of this performance edge must be due to the use of fixed-precision integer arithmetic rather than some multi-precision package such as *gmp* which is used by Yices and CVC3. Also, the goal slices typically have a simple propositional structure and it seems that relatively few goals involve instantiating quantifiers which brings in more propositional structure.

Soundness.

The use of fixed-precision 32-bit arithmetic by Simplify with little or no overflow checking is rather alarming from a soundness point of view. For example, Simplify will happily prove:

```
(IMPLIES
    (EQ x 2000000000)
    (EQ (+ x x) (- 294967296))
)
```

As mentioned earlier, an attempt to soften the impact of this soundness when Simplify was used with ESC/Java involved replacing all integer constants with magnitude above a threshold by symbolic constants. When we tried this approach with a threshold of 100,000, several examples of false goal slices were certified 'true' by Simplify. These particular goals became unproven with a slightly lower threshold of 50,000.

One indicator of when overflow is happening is when Simplify aborts because of an assertion failure. All the reported errors in the Simplify runs are due to failure of an assertion checking that an integer input to a function is positive. We guess this is due to silent arithmetic overflow like in the above example. We investigated how low a threshold was needed for eliminating the errors with the Simulator VCs and found all errors did not go away until we reduced the threshold to 500.

To get a handle on the impact of using a threshold on provability, we reran the Yices test on the Simulator example using various thresholds. With 100,000 the fraction of goals proven by Yices dropped to 90.8%, with 500 to 90.4% and with 20 to 89.6%. Since Yices rejects any additional hypotheses or conclusions which are made non-linear by the introduction of symbolic versions of integer constants, these results indicate that under 2% of the Simulator goal slices involve linear arithmetic problems with multiplication by constants greater than 20.

Timeouts.

To enable working through large sets of problems in reasonable times, it is very useful to be able to interrupt runs after a controllable interval. We found the resource limit of CVC3 allowed fairly reliable stopping of code.

The Yices developers recommended an approach involving using timer interrupts and setting a certain variable in the interrupt handler. However, this feature depended on using an alternate API to the one we used, and this alternate API did not support the creation of quantified formulae. Fortunately, we have not yet needed a timeout feature with our runs of Yices.

We did implement a timer-driven interrupt feature for stopping subprocesses which might be useful for stopping Simplify at some in the future.

Robustness.

When working with an earlier version of CVC3, we had significant problems with it generating segmentation faults and diverging. Because of our interface to CVC3 through its API, every fault would bring down our iteration through the goal slices of one of the examples. We resorted to a tedious process of recording goal slices to be excluded from runs in a special file, with a new entry manually added to this file after each observed crash or divergence. Fortunately the CVC3 developers are responsive to bug reports.

We have found Yices pleasingly stable: to date we have observed only one case in which it has generated a segmentation fault. (This occurred in an experiment not reported in the figures here.)

One incentive for running provers in a subprocess is that the calling program is insulated from crashes of the subprocess.

Incompleteness.

As a first step towards improving the coverage possible with Yices, CVC3 and Simplify, we added axiomatic rules describing the bounding properties of the integer division

	Yices		CVC3		Simplify	
proven	554	96.2%	554	96.2%	560	97.2%
unproven	22	3.8%	0	0 %	16	2.8%
timeout	0	0 %	12	2.1%	0	0 %
error	0	0 %	10	1.7%	0	0 %
total	576		576		576	

Table 6: Autopilot coverage with div & mod rules

	Yices		CVC3		Simplify	
proven	6216	94.3%	6256	94.9%	6045	91.7%
unproven	379	5.7%	108	1.6%	388	5.9%
timeout	0	0 %	231	3.5%	0	0 %
error	0	0 %	0	0 %	162	2.5%
total	6595		6595		6595	

Table 7: Simulator coverage with div & mod rules

and modulo operators:

$$\forall x, y : \mathbf{Z}.\ 0 < y \ \Rightarrow\ 0 \leq x \bmod y$$
$$\forall x, y : \mathbf{Z}.\ 0 < y \ \Rightarrow\ x \bmod y < y$$
$$\forall x, y : \mathbf{Z}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ y \times (x \div y) \ \leq\ x$$
$$\forall x, y : \mathbf{Z}.\ 0 \leq x \ \wedge\ 0 < y \ \Rightarrow\ x - y \ <\ y \times (x \div y)$$
$$\forall x, y : \mathbf{Z}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ x \ \leq\ y \times (x \div y)$$
$$\forall x, y : \mathbf{Z}.\ x \leq 0 \ \wedge\ 0 < y \ \Rightarrow\ y \times (x \div y) \ <\ x + y$$

Rules characterising $\div$ and mod when their second argument is negative can also be formulated, but these were not needed for our examples.

The coverage obtained with these additional rules is shown in Tables 6 and 7.

The observed total run times of the provers were 15-30% slower than without the additional rules.

The extra goal slices proven nearly all involve integer division with a constant divisor. Such instances of division yield instances of the axioms with linear multiplications which the provers can then work with. Most remaining unproven goal slices that were proved true by the Simplifier involved non-constant divisors or non-linear multiplications. We have experimented a little with adding in axioms involving inequalities and non-linear multiplication, but so far have not had much success. A problem is phrasing the axioms so that the instantiation heuristics, perhaps guided by explicit identification of sub-expressions to use for matching, can work productively.

7. FUTURE WORK

7.1 Near term work with SMT solver interface

One objective in the next month or two is to get the interface code into a state in which we can publically release it.

We expect that the main initial use will be in exploiting the counter-example capability to debug code and specifications. SPARK users would be reluctant to trust direct judgement provided by SMT solvers on goal slice truth. However Praxis's interactive prover has been through some certification process and a worthwhile sub-project would be to translate proof objects output by e.g. CVC3 into commands for the interactive prover. Ellis and Ireland in their work also generated proof scripts for the interactive prover from their proof plans that successfully proved VCs.

Another objective is to establish an automatic means for translating SPARK VCs into the SMT-LIB format. This ought to be straightforward given CVC3's capabilities for dumping problems in this format. This would provide a useful way of augmenting the SMT-LIB with the VCs from SPARK code examples such as the Simulator and Autopilot used in our evaluation.

We also eventually would like to try further examples. A problem is the dearth of medium or large SPARK examples in the public domain. Praxis have access in house to some suitable interesting examples, and we hope through collaboration and site visits to also experiment with these.

7.2 The larger picture

Three areas we are considering exploring are non-linear arithmetic, improved quantifier support and automatic invariant generation. Improvements in these areas would be of significant help in increasing the level of automation of VC proof, especially for VCs coming from typical SPARK applications.

Currently we have identified some major lines of relevant work in each area, but have not yet narrowed down on which approaches would be most productive to pursue.

Below we survey some of the literature we have come across on reasoning in these areas and speculate on architectures we might adopt for a full verification system.

7.2.1 Non-linear arithmetic

We are interested in being able to prove problems involving non-linear arithmetic over both the integers and reals. Arithmetic on the reals is of interest for several reasons. Real problems are easier to decide than integer problems and the kinds of integer arithmetic problems that frequently come up in VCs are often also true over reals. Often algorithms for solving integer problems extend algorithms for real problems. For example, this is the case with integer linear programming and mixed integer real non-linear programming. Real arithmetic is also of interest because it is often used to approximate floating-point arithmetic.

The theory of real closed fields (first order formulae over equalities and inequalities involving polynomials over reals) is decidable and decision procedures involving cylindrical algebraic decomposition are under active development [16]. These procedures have high time complexities and are usually only practical on small problems. As we have not yet identified programs yielding interesting VCs in this class, it is difficult to say whether such procedures could be useful to us.

There is much promising work on incomplete proof techniques for quantifier free problems involving polynomials over real numbers. These techniques are observed to be sufficient for problems that come up in practice that are significantly larger than can be handled by complete techniques. For example, Tiwari has investigated using Gröbner bases [30], Parrilo uses sum of squares decompositions and semi-definite programming (a non-linear extension of linear programming) [26] and the ACL2 theorem prover has extensions to support some non-linear resoning [17].

Akbarpour and Paulson are currently exploring heuristically reducing problems involving functions such as sine, ex-

ponentials and logarithms to real closed field problems [3].

We have not come across specific work addressing reasoning with integer division and modulo operators. Since these operators can be fully characterised in terms of integer addition and multiplication with a few first order axioms, we hope that it might be possible to make significant headway with some appropriate combination of first-order reasoning and reasoning about integer polynomial arithmetic. Any such techniques will almost certainly be heuristic in nature, since the problem of solving equations involving polynomials with integer coefficients (Diophantine equations) is undecidable.

7.2.2 Quantifiers

The support for first-order reasoning in the Simplify prover and SMT solvers such as Yices and CVC3 is certainly very useful, but it falls far short of what automated first-order provers are capable of.

Integrating refutation complete first-order provers with reasoning in specific theories such as integer linear arithmetic is known to be a very hard problem.

A promising new direction in first order theorem proving research is that of applying *instantiation-based methods* [13]. Here the aim is to produce refutation complete procedures for first order logic which work by running a SAT solver on successively larger ground instantiations of first-order problem. Given that SMT solvers also use SAT solver algorithms at heart, a natural question that several have asked is whether instance-based and SMT algorithms could be fruitfully combined. Such a combination may well be substantially incomplete, but could still be very useful in practice.

7.2.3 Invariant generation

The need and opportunities for automatic support in the inference of loop invariants have long been recognised [32]. In the last decade there has been a revival in interest in the problem [7, 31].

Flanagan and Leino [11] demonstrate a lightweight generate and test approach for the ESC/Java system: a large number of candidate annotations are generated, inspired by the program structure, and the VC prover then prunes these down.

The technique of *predicate abstraction* [14], a form of abstract interpretation, has been very useful in software model checkers such as Microsoft's SLAM [4] and Berkeley's BLAST [15]. Flanagan and Qadeer [12] explain how to use predicate abstraction to generate loop invariants. An interesting feature of their work is the ability to infer loop invariants with quantifiers, something often necessary when verifying programs involving arrays.

Leino and Logozzo [20] use failure of the VC prover to guide the refinement of conjectured loop invariants just for those program executions associated with the failure. Whereas this work employs abstract interpretation, McMillan [22] achieves a somewhat similar end through the use of Craig interpolants.

Nearly all the above cited work focusses on invariants involving only linear arithmetic expressions. Recently Gröbner basis techniques have been used for handling polynomial arithmetic expressions over the reals [28, 19].

As mentioned in the introduction, Ellis and Ireland [18] have used proof-planning to identify how to strengthen loop invariants.

7.2.4 Verification system architectures

How can different approaches to proving VCs be successfully integrated? One approach is to use a theorem proving environment for programming strategies for refining proof goals and for interfacing to individual provers such as SMT provers, non-linear arithmetic provers and first-order provers. Theorem proving environments allow for rapid experimentation with proof strategies and already have many individual provers of interest either linked in or built in. This approach of using a programmable theorem proving environment as a hub prover was advocated in the PROSPER project [23] and is employed in the Forte system at Intel [29]. Theorem proving environments that look appealing for such a role include Isabelle [25], HOL, HOL Light and PVS.

A finer grain approach might be to investigate adding new procedures as extra theory solvers within an SMT solver. For example, maybe a non-linear arithmetic procedure could be integrated into an SMT solver.

Exploring techniques for invariant generation will require program analysis capabilities and collaboration between a variety of different reasoning tools. SRI have proposed an *evidential tool bus* as an architecture for linking together verification components [27]. They have expressed a specific interest in it being used for generating invariants and supporting software verification. It would be very interesting if we could make use of this and possibly assist in its development.

8. CONCLUSIONS

We have presented some preliminary encouraging results in using the state-of-the-art SMT solvers Yices and CVC3 and the Simplifier prover to discharge verification conditions arising from SPARK programs.

Around 90% of the problems we examined involved no non-linear arithmetic reasoning and were usually solved in under 0.1s by all tools. Another 3-7% were solvable when simple axioms were added concerning bounding properties of the modulo and integer division operators. Many of the remaining true problems are of a slightly more essential non-linear character and are beyond what is provable by the tools even with these axioms. However these problems are mostly still easy to see true, and Praxis's Simplifier prover appears able to handle them in most cases.

We expect shortly to publically release the code we have developed so SPARK users can experiment with it. Our code will also soon provide an easy way of producing SMT challenge problems in the standard SMT-LIB format from SPARK program VCs.

Longer term, we see this work as a first step in a research programme to improve the level of automation in the formal verification of programs written in SPARK and SPARK like subsets of other programming languages.

9. REFERENCES

[1] CVC3: an automatic theorem prover for Satisfiability Modulo Theories (SMT). Homepage at `http://www.cs.nyu.edu/acsys/cvc3/`.

[2] ESC/Java2: Extended Static Checker for Java version 2. Development coordinated by KindSoftware at University College Dublin. Homepage at `http://secure.ucd.ie/products/opensource/ESCJava2/`.

[3] B. Akbarpour and L. C. Paulson. Towards automatic proofs of inequalities involving elementary functions. In *PDPAR: Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2006.

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and Static Driver Verifier: technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft Research, 2004. The SDV homepage is `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`.

[5] J. Barnes. *High Integrity Software: The SPARK approach to safety and security*. Addison Wesley, 2003.

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Post workshop proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*. Springer, 2004.

[7] S. Bensalem and Y. Lakhnech. Automatic generation of invariants. *Formal Methods in System Design*, 15:75–92, 1999.

[8] E. Denney, B. Fischer, and J. Schumann. An empirical evaluation of automated theorem provers in software certification. *International Journal of AI tools*, 15(1):81–107, 2006.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for proof checking. *Journal of the ACM*, 52(3):365–473, 2005.

[10] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at `http://yices.csl.sri.com/tool-paper.pdf`, August 2006.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME: International Symposium of Formal Methods Europe*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL: Principles of Programming Languages*, pages 191–202. ACM, 2002.

[13] H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *LICS: Logic in Computer Science*, pages 55–64. IEEE, 2003.

[14] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV: Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *SPIN: workshop on model checking software*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[16] H. Hong and C. Brown. QEPCAD: Quantifier elimination by partial cylindrical algebraic decomposition, 2004. See `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html` for current implementation.

[17] W. A. Hunt, Jr., R. B. Krug, and J. Moore. Linear and nonlinear arithmetic in ACL2. In *CHARME: Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2003.

[18] A. Ireland, B. Ellis, A. Cook, R. Chapman, and J. Barnes. An integrated approach to high integrity software verification. *Journal of Automated Reasoning*, 36(4):379–410, 2006.

[19] L. I. Kovács and T. Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In *SYNASC: Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 2005.

[20] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS: Programming Languages and Systems, Third Asian Symposium*, volume 3780 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2005.

[21] K. R. M. Leino, J. Saxe, C. Flanagan, J. Kiniry, et al. The logics and calculi of ESC/Java2, revision 1.10. Technical report, University College Dublin, November 2004. Available from the documentation section of the ESC/Java2 web pages.

[22] K. L. McMillan. Lazy abstraction with interpolants. In *CAV: Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[23] T. F. Melham. PROSPER: an investigation into software architecture for embedded proof engines. In *FRoCoS: Frontiers of Combining Systems*, volume 2309 of *Lecture Notes in Artificial Intelligence*, pages 193–206. Springer, 2002.

[24] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on programming Languages and Systems*, 1(2):245–257, 1979.

[25] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. See `http://www.cl.cam.ac.uk/research/hvg/Isabelle/` for current information.

[26] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96(2):293–320, 2003.

[27] J. Rushby. Harnessing disruptive innovation in formal verification. In *SEFM: Software Engineering and Formal Methods*. IEEE, 2006.

[28] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Non-linear loop invariant generation using Gröbner bases. In *POPL: Principles of Programming Languages*, pages 318–329. ACM, 2004.

[29] C.-J. Seger, R. B. Jones, J. W. O'Leary, T. Melham, M. D. Aagaard, C. Barrett, and D. Syme. An industrially effective environment for formal hardware verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(9):1381–1405, 2005.

[30] A. Tiwari. An algebraic approach for the unsatisfiability of nonlinear constraints. In *CSL: Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2005.

[31] A. Tiwari, H. Rueß, H. Saïdi, and N. Shankar. A technique for invariant generation. In *TACAS: tools and algorithms for the construction and analysis of systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2001.

[32] B. Wegbreit. The synthesis of loop predicates. *Communications of the ACM*, 17(2), 1974.

[33] L. Zhang and S. Malik. The quest for efficient boolean satisfiability solvers. In *CAV: Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 17–36. Springer, 2002.

SMT-Based Synthesis of Distributed Systems*

Bernd Finkbeiner
Universität des Saarlandes
finkbeiner@uni-sb.de

Sven Schewe
Universität des Saarlandes
schewe@uni-sb.de

ABSTRACT

We apply SMT solving to synthesize distributed systems from specifications in linear-time temporal logic (LTL). The LTL formula is translated into an equivalent universal co-Büchi tree automaton. The existence of a finite transition system in the language of the automaton is then specified as a quantified formula in the theory $(\mathbb{N}, <)$ of the ordered natural numbers with uninterpreted function symbols. While our experimental results indicate that the resulting satisfiability problem is generally out of reach for the currently available SMT solvers, the problem immediately becomes tractable if we fix an upper bound on the number of states in the distributed system. After replacing each universal quantifier by an explicit conjunction, the SMT solver Yices solves simple single-process synthesis problems within a few seconds, and distributed synthesis problems, such as a two-process distributed arbiter, within one minute.

1. INTRODUCTION

Synthesis automatically *derives* correct implementations from specifications. Compared to verification, which only *proves* that a given implementation is correct, this has the advantage that there is no need to manually write and debug the code.

For temporal logics, the synthesis problem has been studied in several variations, including the synthesis of *closed* and *single-process* systems [2, 12, 6, 7], pipeline and ring architectures [9, 8, 11], as well as general *distributed* architectures [4]. Algorithms for synthesizing distributed systems typically reduce the synthesis problem in a series of automata transformations to the non-emptiness problem of a tree automaton. Unfortunately, the transformations are expensive: for example, in a pipeline architecture, each process requires a powerset construction and therefore causes an exponential blow-up in the number of states.

Inspired by the success of bounded model checking [3, 1], we recently proposed an alternative approach based on a reduction of the synthesis problem to a satisfiability problem [10]. Our starting point is the representation of the LTL specification as a universal co-Büchi tree automaton. The acceptance of a finite-state transition system by a universal co-Büchi automaton can be characterized by the existence of an annotation that maps each pair of a state of the automaton and a state of the transition system to a natural number. We define a constraint system that specifies the existence of a valid annotation and, additionally, ensures that the resulting implementation is consistent with the limited information available to the distributed processes. For this purpose, we introduce a mapping that decomposes the states of the global transition system into the states of the individual processes: because the reaction of a process only depends on its local state, the process is forced to give the same reaction whenever it cannot distinguish between two paths in the global transition system.

In this paper, we report on preliminary experience applying the new approach with the SMT solver Yices. The result of the reduction is a quantified formula in the theory $(\mathbb{N}, <)$ of the ordered natural numbers with uninterpreted function symbols. The formula contains only a single quantifier (over the states of the implementation, represented as natural numbers).

While our experimental results indicate that proving the satisfiability of the quantified formulas is currently not possible (Yices reports "unknown"), the problem immediately becomes tractable if we fix an upper bound on the number of states. After replacing each universal quantifier by an explicit conjunction, Yices solves simple single-process synthesis problems within a few seconds, and distributed synthesis problems, such as a two-process distributed arbiter, within one minute.

2. PRELIMINARIES

We consider the synthesis of distributed reactive systems that are specified in linear-time temporal logic (LTL). Given an architecture A and an LTL formula φ, we determine whether there is an implementation for each system process in A, such that the composition of the implementations satisfies φ.

*This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

AFM'07, November 6, Atlanta, GA, USA.

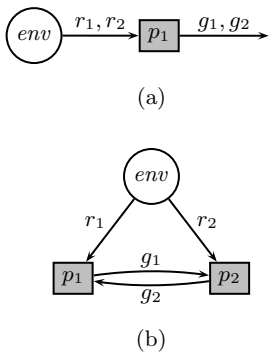


Figure 1: Example architectures: (a) single-process arbiter (b) two-process arbiter

2.1 Architectures

An *architecture* A is a tuple (P, env, V, I, O), where P is a set of processes consisting of a designated environment process $env \in P$ and a set of system processes $P^- = P \setminus \{env\}$. V is a set of boolean system variables (which also serve as atomic propositions), $I = \{I_p \subseteq V \mid p \in P^-\}$ assigns a set I_p of input variables to each system process $p \in P^-$, and $O = \{O_p \subseteq V \mid p \in P\}$ assigns a set O_p of output variables to each process $p \in P$ such that $\bigcup_{p \in P} O_p = V$. While the same variable $v \in V$ may occur in multiple sets in I to indicate broadcasting, the sets in O are assumed to be pairwise disjoint.

Figure 1 shows two example architectures, a single-process arbiter and a two-process arbiter. In the architecture in Figure 1a, the arbiter is a single process (p_1), which receives requests (r_1, r_2) from the environment (env) and reacts by sending grants (g_1, g_2). In the architecture in Figure 1b, the arbiter is split into two processes (p_1, p_2), which each receive one type of request (p_1 receives r_1; p_2 receives r_2) and react by sending the respective grant (p_1 sends g_1; p_2 sends g_2).

2.2 Implementations

We represent implementations as labeled transition systems. For a given finite set Υ of directions and a finite set Σ of labels, a Σ-labeled Υ-*transition system* is a tuple $\mathcal{T} = (T, t_0, \tau, o)$, consisting of a set of states T, an initial state $t_0 \in T$, a transition function $\tau : T \times \Upsilon \to T$, and a labeling function $o : T \to \Sigma$. $\mathcal{T}$ is a *finite-state* transition system iff T is finite.

Each system process $p \in P^-$ is implemented as a 2^{O_p}-labeled 2^{I_p}-transition system $\mathcal{T}_p = (T_p, t_p, \tau_p, o_p)$. The specification φ refers to the composition of the system processes, which is the 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T}_A = (T, t_0, \tau, o)$, defined as follows: the set $T = \bigotimes_{p \in P^-} T_p \times 2^{O_{env}}$ of states is formed by the product of the states of the process transition systems and the possible values of the output variables of the environment. The initial state t_0 is formed by the initial states t_p of the process transition systems and a designated *root direction* $\subseteq O_{env}$. The transition function updates, for each system process p, the T_p part of the state in accordance with the transition function τ_p, using (the projection of) o as input, and updates the $2^{O_{env}}$ part of the state with the output of the environment process. The labeling function o labels each state with the union of its $2^{O_{env}}$ part with the labels of its T_p parts.

With respect to the system processes, the combined transition system thus simulates the behavior of all process transition systems; with respect to the environment process, it is *input-preserving*, i.e., in every state, the label accurately reflects the input received from the environment.

2.3 Synthesis

A specification φ is (finite-state) *realizable* in an architecture $A = (P, V, I, O)$ iff there exists a family of (finite-state) implementations $\{\mathcal{T}_p \mid p \in P^-\}$ of the system processes, such that their composition $\mathcal{T}_A$ satisfies φ.

2.4 Bounded Synthesis

We introduce bounds on the size of the process implementations and on the size of the composition. Given an architecture $A = (P, V, I, O)$, a specification φ is *bounded realizable* with respect to a family of bounds $\{b_p \in \mathbb{N} \mid p \in P^-\}$ on the size of the system processes and a bound $b_A \in \mathbb{N}$ on the size of the composition T_A, if there exists a family of implementations $\{\mathcal{T}_p \mid p \in P^-\}$, where, for each process $p \in P$, $\mathcal{T}_p$ has at most b_p states, such that the composition $\mathcal{T}_A$ satisfies φ and has at most b_A states.

2.5 Tree Automata

An *alternating parity tree automaton* is a tuple $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$, where Σ denotes a finite set of labels, Υ denotes a finite set of directions, Q denotes a finite set of states, $q_0 \in Q$ denotes a designated initial state, δ denotes a transition function, and $\alpha : Q \to C \subset \mathbb{N}$ is a coloring function. The transition function $\delta : Q \times \Sigma \to \mathbb{B}^+(Q \times \Upsilon)$ maps a state and an input letter to a positive boolean combination of states and directions. In our setting, the automaton runs on Σ-labeled Υ-transition systems. The acceptance mechanism is defined in terms of run graphs.

A *run graph* of an automaton $\mathcal{A} = (\Sigma, \Upsilon, Q, q_0, \delta, \alpha)$ on a Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is a minimal directed graph $\mathcal{G} = (G, E)$ that satisfies the following constraints:

- The vertices $G \subseteq Q \times T$ form a subset of the product of Q and T.
- The pair of initial states $(q_0, t_0) \in G$ is a vertex of $\mathcal{G}$.
- For each vertex $(q, t) \in G$, the set $\{(q', \upsilon) \in Q \times \Upsilon \mid ((q, t), (q', \tau(t, \upsilon))) \in E\}$ satisfies $\delta(q, o(t))$.

A run graph is *accepting* if every infinite path $g_0 g_1 g_2 \ldots \in G^\omega$ in the run graph satisfies the *parity condition*, which requires that the highest number occurring infinitely often in the sequence $\alpha_0 \alpha_1 \alpha_2 \in \mathbb{N}$ with $\alpha_i = \alpha(q_i)$ and $g_i = (q_i, t_i)$ is even. A transition system is accepted if it has an accepting run graph.

The set of transition systems accepted by an automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty iff its

language is empty. An alternating automaton is called *universal* if, for all states q and input letters σ, $\delta(q, \sigma)$ is a conjunction.

A parity automaton is called a *Büchi* automaton if the image of α is contained in $\{1, 2\}$ and a *co-Büchi* automaton iff the image of α is contained in $\{0, 1\}$. Büchi and co-Büchi automata are denoted by $(\Sigma, \Upsilon, Q, q_0, \delta, F)$, where $F \subseteq Q$ denotes the states with the higher color. A run graph of a Büchi automaton is thus accepting if, on every infinite path, there are infinitely many visits to F; a run graph of a co-Büchi automaton is accepting if, on every path, there are only finitely many visits to F.

3. ANNOTATED TRANSITION SYSTEMS

In this section, we discuss an annotation function for transition systems. The annotation function has the useful property that a finite-state transition system satisfies the specification if and only if it has a valid annotation.

Our starting point is a representation of the specification as a universal co-Büchi automaton.

THEOREM 1. *[5] Given an LTL formula φ, we can construct a universal co-Büchi automaton $\mathcal{U}_\varphi$ with $2^{O(|\varphi|)}$ states that accepts a transition system $\mathcal{T}$ iff $\mathcal{T}$ satisfies φ.* □

An *annotation* of a transition system $\mathcal{T} = (T, t_0, \tau, o)$ on a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ is a function $\lambda : Q \times T \to \{\_\} \cup \mathbb{N}$. We call an annotation *c-bounded* if its mapping is contained in $\{\_\} \cup \{0, \ldots, c\}$, and *bounded* if it is c-bounded for some $c \in \mathbb{N}$. An annotation is *valid* if it satisfies the following conditions:

1. the pair (q_0, t_0) of initial states is annotated with a natural number ($\lambda(q_0, t_0) \neq \_$), and
2. if a pair (q, t) is annotated with a natural number ($\lambda(q, t) = n \neq \_$) and $(q', \upsilon) \in \delta(q, o(t))$ is an atom of the conjunction $\delta(q, o(t))$, then $(q', \tau(t, \upsilon))$ is annotated with a greater number, which needs to be strictly greater if $q' \in F$ is rejecting. That is, $\lambda(q', \tau(t, \upsilon)) \triangleright_{q'} n$ where $\triangleright_{q'}$ is $>$ for $q' \in F$ and $\geq$ otherwise.

THEOREM 2. *[10] A finite-state Σ-labeled Υ-transition system $\mathcal{T} = (T, t_0, \tau, o)$ is accepted by a universal co-Büchi automaton $\mathcal{U} = (\Sigma, \Upsilon, Q, \delta, F)$ iff it has a valid $(|T| \cdot |F|)$-bounded annotation.*

4. SINGLE-PROCESS SYNTHESIS

Using the annotation function, we reduce the non-emptiness problem of the universal co-Büchi tree automaton to an SMT problem. This solves the synthesis problem for single-process systems.

We represent the (unknown) transition system and its annotation by uninterpreted functions. The existence of a valid annotation is thus reduced to the satisfiability of a constraint system in first-order logic modulo finite integer arithmetic. The advantage of this representation is that the size of the constraint system is small (bilinear in the size of $\mathcal{U}$ and the number of directions). Furthermore, the additional constraints needed for distributed synthesis, which will be defined in Section 5, have a likewise compact representation.

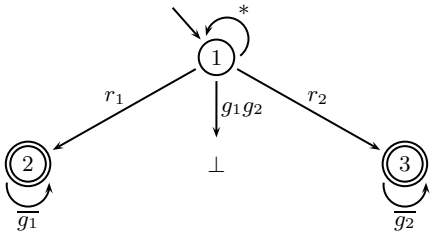


Figure 2: Specification of a simple arbiter, represented as a universal co-Büchi automaton. The states depicted as double circles (2 and 3) are the rejecting states in F.

The constraint system specifies the existence of a finite input-preserving 2^V-labeled $2^{O_{env}}$-transition system $\mathcal{T} = (T, t_0, \tau, o)$ that is accepted by the universal co-Büchi automaton $\mathcal{U}_\varphi = (\Sigma, \Upsilon, Q, q_0, \delta, F)$ and has a valid annotation λ.

To encode the transition function τ, we introduce a unary function symbol τ_υ for every output $\upsilon \subseteq O_{env}$ of the environment. Intuitively, τ_υ maps a state t of the transition system $\mathcal{T}$ to its υ-successor $\tau_\upsilon(t) = \tau(t, \upsilon)$.

To encode the labeling function o, we introduce a unary predicate symbol a for every variable $a \in V$. Intuitively, a maps a state t of the transition system $\mathcal{T}$ to *true* iff it is part of the label $o(t) \ni a$ of $\mathcal{T}$ in t.

To encode the annotation, we introduce, for each state q of the universal co-Büchi automaton $\mathcal{U}$, a unary predicate symbol $\lambda_q^{\mathbb{B}}$ and a unary function symbol $\lambda_q^{\#}$. Intuitively, $\lambda_q^{\mathbb{B}}$ maps a state t of the transition system $\mathcal{T}$ to *true* iff $\lambda(q, t)$ is a natural number, and $\lambda_q^{\#}$ maps a state t of the transition system $\mathcal{T}$ to $\lambda(q, t)$ if $\lambda(q, t)$ is a natural number and is unconstrained if $\lambda(q, t) = \_$.

We can now formalize that the annotation of the transition system is valid by the following first order *progress* constraints (modulo finite integer arithmetic):
$\forall t.\, \lambda_q^{\mathbb{B}}(t) \;\wedge\; \underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))} \;\rightarrow\; \lambda_{q'}^{\mathbb{B}}(\tau_\upsilon(t)) \;\wedge\; \lambda_{q'}^{\#}(\tau_\upsilon(t)) \triangleright_q \lambda_q^{\#}(t)$, where $\overrightarrow{a}(t)$ represents the label $o(t)$, $\underline{(q', \upsilon) \in \delta(q, \overrightarrow{a}(t))}$ represents the corresponding propositional formula, and $\triangleright_q$ stands for $\triangleright_q \equiv >$ if $q \in F$ and $\triangleright_q \equiv \geq$ otherwise. Additionally, we require the *initiality constraint* $\lambda_{q_0}^{\mathbb{B}}(0)$, i.e., we require the pair of initial states to be labeled by a natural number (w.l.o.g. $t_0 = 0$).

To guarantee that the resulting transition system is input-preserving, we add, for each $a \in O_{env}$ and each $\upsilon \subseteq O_{env}$, a *global consistency* constraint $\forall t.\, a(\tau_\upsilon(t))$ if $a \in \upsilon$ and $\forall t.\, \neg a(\tau_\upsilon(t))$ if $a \notin \upsilon$. Additionally, we require the *root constraint* that the initial state is labeled with the root direction.

Example. Consider the specification of a simple arbiter, depicted as a universal co-Büchi automaton in Figure 2. The specification requires that globally (1) at most one process has a grant and (2) each request is eventually followed by a grant.

Figure 3 shows the constraint system, resulting from the specification of an arbiter by the universal co-Büchi automaton depicted in Figure 2, implemented as a single process as required by the architecture of Figure 1a.

The first constraint represents the requirement that the resulting transition system must be input-preserving, the second requirement represents the initialization (where $\neg r_1(0) \wedge \neg r_2(0)$ represents an arbitrarily chosen root direction), and the requirements 3 through 8 each encode one transition of the universal automaton of Figure 2. Following the notation of Figure 2, r_1 and r_2 represent the requests and g_1 and g_2 represent the grants.

5. DISTRIBUTED SYNTHESIS

To solve the distributed synthesis problem for a given architecture $A = (P, V, I, O)$, we need to find a family of (finite-state) transition systems $\{\mathcal{T}_p = (T_p, t_0^p, \tau_p, o_p) \mid p \in P^-\}$ such that their composition to $\mathcal{T}_A$ satisfies the specification. The constraint system developed in the previous section can be adapted to distributed synthesis by explicitly decomposing the global state space of the combined transition system $\mathcal{T}_A$: we introduce a unary function symbol d_p for each process $p \in P^-$, which, intuitively, maps a state $t \in T_A$ of the product state space to its p-component $t_p \in T_p$.

The value of an output variable $a \in O_p$ may only depend on the state of the process transition system $\mathcal{T}_p$. We therefore replace every occurrence of $a(t)$ in the constraint system of the previous section by $a(d_p(t))$. Additionally, we require that every process p acts consistently on any two histories that it cannot distinguish. The update of the state of $\mathcal{T}_p$ may thus only depend on the state of $\mathcal{T}_p$ and the input visible to p.

The input visible to p consists of the fragment $I_p^{env} = O_{env} \cap I_p$ of environment variables visible to p, and the set $I_p^{sys} = I_p \setminus O_{env}$ of system variables visible to p. To encode the transition function τ_p, we introduce a $|I_p^{sys}|+1$-ary function symbol τ_p^{υ} for every $\upsilon \subseteq I_p^{env}$. Intuitively, τ_p^{υ} maps the visible input $\upsilon' \subseteq I_p^{sys}$ of the system and a local position l of the transition system $\mathcal{T}_p$ to the $\upsilon \cup \upsilon'$-successor $\tau_p(l, \upsilon \cup \upsilon') = \tau_p^{\upsilon}(\upsilon', l)$ of l. This is formalized by the following *local consistency* constraints:

> $\forall t.\, \tau_p^{\upsilon}(a_1(d_{q_1}(t)), \ldots, a_n(d_{q_n}(t)); d_p(t)) = d_p(\tau_{\upsilon'}(t))$
> for all decisions $\upsilon' \subseteq O_{env}$ of the environment and their fragment $\upsilon = \upsilon' \cap I_p$ visible to p, where the variables $a_1, \ldots, a_n$ form the elements of I_p^{sys}.

Since the combined transition system $\mathcal{T}_A$ is finite-state, the satisfiability of this constraint system modulo finite integer arithmetic is equivalent to the distributed synthesis problem.

Example. As an example for the reduction of the distributed synthesis problem to SMT, we consider the problem of finding a distributed implementation to the arbiter specified by the universal automaton of Figure 2 in the architecture of Figure 1b. The functions d_1 and d_2 are the mappings to the processes p_1 and p_2, which receive requests r_1 and r_2 and provide grants g_1 and g_2, respectively. Figure 4 shows the resulting constraint system. Constraints 1–3, 5, and 6 are the same as in the fully informed case (Figure 3). The consistency constraints 9–10 guarantee that processes p_1 and p_2 show the same behavior on all input histories they cannot distinguish.

1. $\forall t.\, r_1(\tau_{r_1 r_2}(t)) \wedge r_2(\tau_{r_1 r_2}(t)) \wedge r_1(\tau_{r_1 \overline{r}_2}(t))$
   $\wedge \neg r_2(\tau_{r_1 \overline{r}_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 r_2}(t))$
   $\wedge r_2(\tau_{\overline{r}_1 r_2}(t)) \wedge \neg r_1(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \neg r_2(\tau_{\overline{r}_1 \overline{r}_2}(t))$
2. $\lambda_1^{\mathbb{B}}(0) \wedge \neg r_1(0) \wedge \neg r_2(0)$
3. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{\overline{r}_1 r_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 \overline{r}_2}(t)) \geq \lambda_1^{\#}(t)$
   $\wedge\ \lambda_1^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_1^{\#}(\tau_{r_1 r_2}(t)) \geq \lambda_1^{\#}(t)$
4. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(t) \vee \neg g_2(t)$
5. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_1(t) \rightarrow$
   $\lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$
6. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \wedge r_2(t) \rightarrow$
   $\lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_1^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_1^{\#}(t)$
7. $\forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(t) \rightarrow$
   $\lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_2^{\#}(t)$
   $\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t)$
8. $\forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(t) \rightarrow$
   $\lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 \overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 \overline{r}_2}(t)) > \lambda_3^{\#}(t)$
   $\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)$

Figure 3: Example of a constraint system for the synthesis of a single-process system. The figure shows the constraint system for the arbiter example (Figure 2). The arbiter is to be implemented as a single process as shown in Figure 1a.

6. EDGE-BASED ACCEPTANCE

A variation of our construction is to start with a tree automaton that has an edge-based acceptance condition instead of the standard state-based acceptance condition of the automata of Theorem 1. Since the progress constraints refer to edges rather than states, this often leads to a significant reduction in the size of the constraint system.

4. $\forall t.\, \lambda_1^{\mathbb{B}}(t) \rightarrow \neg g_1(d_1(t)) \vee \neg g_2(d_2(t))$

7. $\forall t.\, \lambda_2^{\mathbb{B}}(t) \wedge \neg g_1(d_1(t)) \rightarrow$
$$\lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_2^{\#}(t)$$
$$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_2^{\#}(t)$$
$$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_2^{\#}(t)$$
$$\wedge\ \lambda_2^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_2^{\#}(\tau_{r_1 r_2}(t)) > \lambda_2^{\#}(t)$$

8. $\forall t.\, \lambda_3^{\mathbb{B}}(t) \wedge \neg g_2(d_2(t)) \rightarrow$
$$\lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1\overline{r}_2}(t)) > \lambda_3^{\#}(t)$$
$$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{\overline{r}_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{\overline{r}_1 r_2}(t)) > \lambda_3^{\#}(t)$$
$$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1\overline{r}_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1\overline{r}_2}(t)) > \lambda_3^{\#}(t)$$
$$\wedge\ \lambda_3^{\mathbb{B}}(\tau_{r_1 r_2}(t)) \wedge \lambda_3^{\#}(\tau_{r_1 r_2}(t)) > \lambda_3^{\#}(t)$$

9. $\forall t.\, \tau_1^{r_1}\big(g_2(d_2(t)), d_1(t)\big) = d_1(\tau_{r_1 r_2}(t)) = d_1(\tau_{r_1\overline{r}_2}(t))$
$\wedge\ \tau_1^{\overline{r}_1}\big(g_2(d_2(t)), d_1(t)\big) = d_1(\tau_{\overline{r}_1 r_2}(t)) = d_1(\tau_{\overline{r}_1\overline{r}_2}(t))$

10. $\forall t.\, \tau_2^{r_2}\big(g_1(d_1(t)), d_2(t)\big) = d_2(\tau_{r_1 r_2}(t)) = d_2(\tau_{\overline{r}_1 r_2}(t))$
$\wedge\ \tau_2^{\overline{r}_2}\big(g_1(d_1(t)), d_2(t)\big) = d_2(\tau_{r_1\overline{r}_2}(t)) = d_2(\tau_{\overline{r}_1\overline{r}_2}(t))$

Figure 4: Example of a constraint system for distributed synthesis. The figure shows modifications and extensions to the constraint system from Figure 3 for the arbiter example (Figure 2) in order to implement the arbiter in the distributed architecture shown in Figure 1b.

For universal automata, the transition function δ can be described as a set of edges $E_\delta \subseteq Q \times \Sigma \times Q \times \Upsilon$ with

$$e = (q, \sigma, q', \upsilon) \in E_\delta \Leftrightarrow (q', \upsilon) \text{ is a conjunct of } \delta(q, \sigma).$$

For an edge-based universal co-Büchi automaton $\mathcal{E} = (\Sigma, \Upsilon, Q, E, F)$, the acceptance is defined by a finite set $F \subseteq E$ of rejecting edges, and $\mathcal{E}$ accepts an input tree if all paths in the run graph contain only finitely many rejecting edges. A state-based acceptance condition can be viewed as a special case of an edge-based acceptance condition, where an edge is rejecting iff it originates from a rejecting state, and edge-based acceptance can be translated into state-based acceptance by splitting the states with outgoing accepting and rejecting edges. For an edge-based universal co-Büchi automaton $\mathcal{E}$, we only need to adjust the definition of valid annotations slightly to

2. if a pair (q, t) is annotated with a natural number ($\lambda(q,t) = n \neq \_$) and $(q, o(t), q', \upsilon) = e \in E$ is an edge of $\mathcal{E}$, then $(q', \tau(t, \upsilon))$ is annotated with a greater number, which needs to be strictly greater if $e \in F$ is rejecting. That is, $\lambda(q', \tau(t, \upsilon)) \triangleright_e n$ where $\triangleright_e$ is $>$ for $e \in F$ and $\geq$ otherwise.

Example. Figure 5 shows an example of a universal co-Büchi word automaton with edge-based acceptance condition. The automaton extends the specification of the simple arbiter such that the arbiter may not withdraw a grant while the environment upholds the request. Nonstarvation is required whenever the grant is not kept forever by the other process. Describing the same property with a state-based acceptance conditions requires 40% more states.

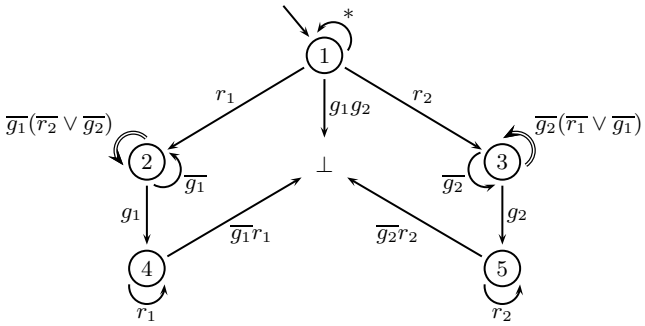


Figure 5: Extended specification of an arbiter, represented as a universal co-Büchi automaton with edge-based acceptance. The edges depicted as double-line arrows are the rejecting edges in F.

7. EXPERIMENTAL RESULTS

Using the reduction described in the previous sections, we considered five benchmarks; we synthesized implementations for simple arbiter specification from Figure 2 and the two architectures from Figure 1, and for a full arbiter specification and the two architectures from Figure 1, and we synthesized a strategy for dining philosophers to satisfy the specification from Figure 6. The arbiter examples are parameterized in the size of the transition system(s), the dining philosophers benchmark is additionally parameterized in the number of philosopher. As the SMT solver, we used Yices version 1.0.9 on a 2.6 Ghz Opteron system.

In all benchmarks, Yices is unable to directly determine the satisfiability of the quantified formulas. (For example the formulas from Figure 3 and Figure 4, respectively, for the monolithic and distributed synthesis in the simple arbiter example.) However, after replacing the universal quantifiers with explicit conjunctions (for a given upper bound on the number of states in the implementation), Yices solved all satisfiability problems quickly.

A single-process implementation of the arbiter needs 8 states. Table 1 shows the time and memory consumption of Yices when solving the SMT problem from Figure 3 with the quantifiers unravelled for different upper bounds on the number of states. The correct implementation with 8 states is found in 8 seconds.

7.1 Arbiter

Table 2 shows the time and memory consumption for the distributed synthesis problem. The quantifiers in the formula from Figure 4 were unravelled for different bounds on the size of the global transition system and for different bounds (shown in parentheses) on the size of the processes. A correct solution with 8 global states is found by Yices in 71 seconds if the number of process states is left unconstrained. Restricting the process states explicitly to 2 leads to an acceleration by a factor of two (36 seconds).

Table 3 and Table 4 show the time and memory consumption of Yices when solving the SMT problem resulting from the arbiter specification of Figure 5. The correct monolithic im-

bound	4	5	6	7	8	9
result	unsatisfiable	unsatisfiable	unsatisfiable	unsatisfiable	satisfiable	satisfiable
# decisions	3957	13329	23881	68628	72655	72655
# conflicts	209	724	1998	15859	4478	4478
# boolean variables	1011	2486	4169	9904	5214	5214
memory (MB)	16.9102	18.1133	20.168	27.4141	26.4375	26.4414
time (seconds)	0.05	0.28	1.53	35.99	7.53	7.31

Table 1: Experimental results from the synthesis of a single-process arbiter using the specification from Figure 2 and the architecture from Figure 1a. The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Figure 3, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

bound	4	5	6	7	8	9	8 (1)	8 (2)
result	unsatisfiable	unsatisfiable	unsatisfiable	unsatisfiable	satisfiable	satisfiable	unsatisfiable	satisfiable
# decisions	6041	15008	35977	89766	197150	154315	178350	71074
# conflicts	236	929	2954	30454	33496	24607	96961	18263
# boolean variables	1269	2944	5793	9194	7766	8533	12403	6382
memory (MB)	17.0469	18.4766	22.1992	33.1211	37.4297	36.2734	39.4922	29.1992
time (seconds)	0.06	0.35	3.3	120.56	70.97	58.43	200.07	36.38

Table 2: Experimental results from the synthesis of a two-process arbiter using the specification from Figure 2 and the architecture from Figure 1b. The table shows the time and memory consumption of Yices 1.0.9 when solving the SMT problem from Figure 4, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in parentheses).

bound	4	5	6	7	8
result	unsatisfiable	satisfiable	satisfiable	satisfiable	satisfiable
# decisions	17566	30011	52140	123932	161570
# conflicts	458	800	1375	2614	3987
# boolean variables	1850	2854	3734	5406	6319
memory (MB)	18.3008	20.0586	22.5781	27.5000	35.7148
time (seconds)	0.21	0.63	1.72	5.15	12.38

Table 3: Experimental results from the synthesis of a single-process arbiter using the specification from Figure 5 and the architecture from Figure 1a. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

bound	4	5	6	7	8	9	8 (1)	8 (2)	8 (3)	8 (4)
result	unsat	unsat	unsat	unsat	sat	sat	unsat	unsat	sat	sat
# decisions	16725	47600	91480	216129	204062	344244	309700	1122755	167397	208255
# conflicts	326	1422	8310	61010	11478	16347	92712	775573	13086	13153
# boolean variables	1890	7788	5793	13028	8330	10665	15395	25340	8240	7806
memory (MB)	18.0273	22.2109	28.5312	43.8594	42.2344	61.9727	54.1641	120.0160	42.1484	42.7188
time (seconds)	0.16	1.72	14.84	208.78	32.47	72.97	263.44	5537.68	31.12	30.36

Table 4: Experimental results from the synthesis of a two-process arbiter using the specification from Figure 5 and the architecture from Figure 1b. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the global transition system and on the number of states in the individual processes (shown in parentheses).

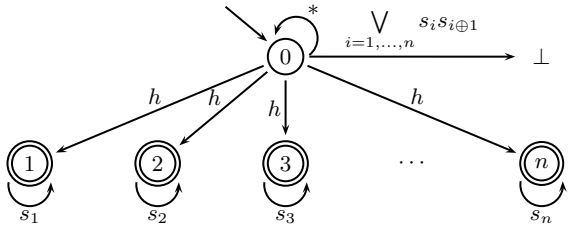


Figure 6: Specification of a dining philosopher problem with n philosophers. The environment can cause the philosophers to become hungry (by setting h to true). The states depicted as double circles (1 through n) are the rejecting states in F; state i refers to the situation where philosopher i is hungry and starving (s_i). The fail state is reached when two adjacent philosophers try to reach for their common chopstick; the fail state refers to the resulting eternal philosophical quarrel that keeps the affected philosophers from eating.

	3 states			4 states			6 states		
# philosophers	time (s)	memory (MB)	result	time (s)	memory (MB)	result	time (s)	memory (MB)	result
125	1.52	23.2695	unsat	23.84	36.2305	unsat	236.5	87.7852	sat
250	5.41	29.2695	unsat	130.07	52.0859	sat	141.36	91.1328	sat
375	22.81	38.9727	unsat	128.83	58.1992	unsat	890.58	154.355	sat
500	17.98	39.9297	unsat	15.84	52.9336	sat	237.04	119.309	sat
625	35.57	49.5586	unsat	417.05	94.7188	unsat	486.5	130.977	sat
750	22.25	52.3359	unsat	20.85	69.1562	sat	82.63	99.707	sat
875	51.98	56.0859	unsat	628.84	119.363	unsat	2546.88	255.965	sat
1000	168.17	70.3906	unsat	734.74	117.703	sat	46.18	124.691	sat
1125	67.14	70.1133	unsat	1555.18	165.922	unsat	1854.77	246.848	sat
1250	165.59	76.2227	unsat	122.8	107.645	sat	596.8	203.012	sat
1375	104.27	75.4531	unsat	3518.85	191.113	unsat	8486.18	490.566	sat
1500	187.25	82.8867	unsat	85.52	129.215	sat	232.81	214.68	sat
1625	85.83	88.8047	unsat	2651.82	246.734	unsat	1437.45	281.203	sat
1750	169.93	97.543	unsat	107.14	126.477	sat	257.77	185.887	sat
1875	174.03	105.25	unsat	3629.18	234.527	unsat	4641.03	405.781	sat
2000	25.86	102.125	unsat	242.55	157.734	sat	811.78	269.375	sat
2125	163.39	113.27	unsat	5932.24	315.711	unsat	6465.75	424.121	sat
2250	412.37	115.438	unsat	523.87	162.391	sat	5034.83	456.316	sat
2375	201.95	120.047	unsat	7311.03	313.168	unsat	4887.76	451.332	sat
2500	375.29	135.535	unsat	235.17	202.59	sat	319.78	253.781	sat
2625	544.03	135.379	unsat	6560.53	312.355	unsat	23990.5	808.633	sat
2750	559.35	139.137	unsat	817.41	226.082	sat	632.28	349.992	sat
2875	308.36	151.727	unsat	7273.89	299.016	unsat	8638.96	551.5	sat
3000	666.18	155.57	unsat	533.23	228.961	sat	3158.26	493.617	sat
3125	235.52	141.93	unsat	12596.6	377.328	unsat	10819.7	693.133	sat
3250	869.53	153.633	unsat	2089.72	308.719	sat	21298.8	889.285	sat
3375	260.88	145.918	unsat	11581.7	379.949	unsat	21560	741.09	sat
3500	308.23	169.348	unsat	897.6	270.676	sat	829.52	398.008	sat
5000	982.68	240.273	unsat	3603.7	421.832	sat	1357.48	582.457	sat
7000	2351.87	313.277	unsat	7069.55	535.98	sat	6438.73	1081.68	sat
10000	4338.83	448.648	unsat	4224.28	761.008	sat	10504.6	1121.58	sat

Table 5: Experimental results from the synthesis of a strategy for the dining philosophers using the specification from Figure 6. The table shows the time and memory consumption of Yices 1.0.9 when solving the resulting SMT problem, with all quantifiers replaced by explicit conjunctions for different bounds on the number of states in the transition system.

plementation with 5 states is found in less than one second, and Yices needs only half a minute to construct a correct distribute implementation. The table also shows that borderline cases like the fruitless search for an implementation with 8 states, but only 2 local states, can become very expensive; in the example, Yices needed more than $1\frac{1}{2}$ hours to determine unsatisfiability. Compromising on optimality, by slightly increasing the bounds, greatly improves the performance. Searching for an implementation with 8 states and 3 local states takes about 30 seconds.

7.2 Dining Philosophers

Table 5 shows the time and memory consumption for synthesizing a strategy for the dining philosophers to satisfy the specification shown in Figure 6. In the dining philosophers benchmark, the size of the specification grows linearly with the number of philosophers; for 10.000 philosophers this results in systems of hundreds of thousands constraints. In spite of the large size of the resulting constraint system, the synthesis problem remains tractable; Yices solves all resulting constraint systems within a few hours, and within a minutes for small constraint systems with up to 1000 philosophers.

8. CONCLUSIONS

Our experimental results suggest that the synthesis problem can be solved efficiently using satisfiability checking as long as a reasonable bound on the size of the implementation can be set in advance. In general, distributed synthesis is undecidable. By iteratively increasing the bound, our approach can be used as a semi-decision procedure.

Bounded synthesis thus appears to be a promising new application domain for SMT solvers. Clearly, there is a lot of potential for improving the performance. For example, Yices is not able to determine the satisfiability of the quantified formula directly. After applying a preprocessing step that replaces universal quantification by explicit conjunctions, Yices solves the resulting satisfiability problem within seconds. Developing specialized quantifier elimination heuristics could be an important step in bringing synthesis to practice.

9. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58:118–149, 2003.

[2] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. IBM Workshop on Logics of Programs*, pages 52–71. Springer-Verlag, 1981.

[3] F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. of CAV*, LNCS. Springer Verlag, 2001.

[4] B. Finkbeiner and S. Schewe. Uniform distributed synthesis. In *Proc. LICS*, pages 321–330. IEEE Computer Society Press, June 2005.

[5] O. Kupferman and M. Vardi. Safraless decision procedures. In *Proc. 46th IEEE Symp. on Foundations of Computer Science*, pages 531–540, Pittsburgh, October 2005.

[6] O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *Proc. ICTL*, pages 91–106, Manchester, July 1997.

[7] O. Kupferman and M. Y. Vardi. μ-calculus synthesis. In *Proc. MFCS*, pages 497–507. Springer-Verlag, 2000.

[8] O. Kupferman and M. Y. Vardi. Synthesizing distributed systems. In *Proc. LICS*, pages 389–398. IEEE Computer Society Press, July 2001.

[9] R. Rosner. *Modular Synthesis of Reactive Systems*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[10] S. Schewe and B. Finkbeiner. Bounded synthesis. In *5th International Symposium on Automated Technology for Verification and Analysis (ATVA 2007)*. Springer Verlag, 2007.

[11] I. Walukiewicz and S. Mohalik. Distributed games. In *Proc. FSTTCS'03*, pages 338–351. Springer-Verlag, 2003.

[12] P. Wolper. *Synthesis of Communicating Processes from Temporal-Logic Specifications*. PhD thesis, Stanford University, 1982.