Generalizing Design Space Exploration with FORMULA

Ethan Jackson
Research in Software Engineering (RiSE), Microsoft Research
Verified Software Workshop, Beijing, China
Introduction to DSE and FORMULA

With Applications to Cyber-Physical Systems
The Next Generation of Applications

Cyber-physical systems (CPS) react to and affect physical systems. CPSs expose software modules to higher-level command-and-control, enabling new features.

Medical Devices
- **Today**
  - Record sensor data
- **Tomorrow**
  - Automatically transmit to e-Health portal through any public wireless network

Aerospace and Defense
- **Today**
  - Remotely explore planets
- **Tomorrow**
  - Autonomous aid human astronauts in dangerous situations

Automotive
- **Today**
  - Real-time traffic information
- **Tomorrow**
  - Use real-time traffic information to drive platoons of vehicles in close formation
Many Challenges Along the Way

Cyber-physical systems must be safe and reliable. They must perform complex computations with few resources.

**Legal Challenges**
How can CPSs be guaranteed safe, at least up to some legal standards?

**Computation and Physics**
How do we predict the behaviors of complex software reacting to physical processes?

**Resource Constraints**
How do we meet functional requirements while satisfying constraints on power, memory, execution time?
Example: Architectural Constraints

Correctly place components on a well-balanced network of ECUs.

Constraints

Tasks in conflict cannot be placed on the same ECU.

An ECU cannot support more than 2 incoming and 2 outgoing communication links.

Every communication link has a capacity; capacities must be balanced.

These “architectural” constraints are in addition to and entangled with implementation details.
The DSE Tool Flow

Model-based development allows architectural problems to be distilled to their essence. Formal methods help the engineer find a suitable design.

1. Requirements elicitation to collect constraints on architectures
2. Formalize design constraints into a domain-specific language.
3. Build a partial system model using your domain-specific language.
4. Let design-space exploration fill in the rest.
The FORUMLA framework allows complex design constraints to be easily specified.

**Functional Decomposition**
Software components are tasks. Two tasks can be in conflict, so they should not be placed on the same device.

```plaintext
1. domain Functionality
2. {
3. Task : (id: Basic).
5. }
```

**Execution Platform**
An interconnection of devices with bounded unidirectional communication channels. Channel capacities must be balanced.

```plaintext
12. domain Distribution
13. {
15. [partial_function]
17. 
19. bigFanIn : ? d is Device,
20. count(Channel(_,d,_)) > 2.
21. bigFanOut : ? d is Device,
22. count(Channel(d,_,_)) > 2.
23. clog : ? d is Device, sum(Channel(_,d,_),2) != sum(Channel(d,_,_),2).
27. }
```
FORMULA provides language level features for correctly building larger specifications. In this example, an architecture must assign tasks to devices correctly.

```
28. domain Architecture
29.  cextends (Functionality * Distribution)
30. {
32.    conflict :: Binding(t1,d), Binding(t2,d),
33.                      Conflict(t1,t2).
34.    conforms :: !conflict.
35. }
```
Design space exploration (DSE) uses specifications to find architectures satisfying all constraints.

The user provides some basic visualization information for viewing models.
Design Space Exploration - Explore

One-click of the exploration button produces candidate architectures.

Automated-layout is used to presents solution in a gallery. A non-graphical description of each model is also generated.
FORMULA is a holistic framework for modeling the next-generation of cyber-physical systems.

Formal Specifications
DSLs, Models, and Model Transformations are formally specified.

Safe Composition
A novel set of composition operators allow formal specifications to be correctly combined.

Formal Analysis
State-of-the-art formal analysis is provided to enable tasks such as design space exploration.
Representing Design Spaces

An Overview of FORMULA Language Features
The FORMULA Approach

Language Level
A novel set of composition operators allow formal specifications to be correctly combined.

Formal Semantics

Formal Methods

Module system produces new specs. With semantic guarantees

DSLs are ADTs + CLP

Transformations are ADTs + CLP

First-order formulas over theories plus fixed-point operator

Finite modeling finding loop over an SMT solver Z3
Domains Encapsulate Abstractions

A domain represents an abstraction through data types and declarative constraints

```
domain Functionality
{
    Task : (id: Basic).
    [relation]
    Conflict : (t1: Task, t2: Task).
}.
```

A “model” is just a set of records instantiated with respect to the data types of some domain and satisfy domain constraints.

```
model Example : Functionality
{
    tAcc = Task("Accelerometer"),
    tFil = Task("Filter"),
    tArB = Task("AirBagActuator"),
    Conflict(tAcc, tFil),
    Conflict(tAcc, tArB),
    Conflict(tFil, tArB)
}.
```

Claims that “Example” is a model of “Functionality”.

Simple constraint annotations
Declarative Constraints

A domain represents an abstraction through data types and declarative constraints.

```c
domain Distribution
{
    Device : (id: Basic).
    [partial_function]
    Channel : (from: Device, to: Device, cap: PosInteger).

    bigFanIn : ? d is Device, count(Channel(_,d,_)) > 2.
    bigFanOut : ? d is Device, count(Channel(d,_,_)) > 2.
    clog : ? d is Device, sum(Channel(_,d,_),2) != sum(Channel(d,_,_),2).
    conforms : !(bigFanIn | bigFanOut | clog).
}
```

Named queries

Constraints

Standardized conformance test

Aggregation operators

Queries are evaluated by first deducing all facts from an input model (forward-chaining) and then trying to find an assignment of variables that show the query to be true.
We have an order sorted set of infinite alphabets closed under intersection:

\[ \tau(c) = \Sigma_k \]

Nil, PosInteger, NegInteger, Natural, Real, String

Free constructors create records; these are equated by structural equality.

\[ f : (T_1, T_2, \ldots, T_n) \]

\[ \tau(f) = \{ f(t_1, t_2, \ldots, t_n) \mid \forall 1 \leq i \leq n, t_i \in \tau(T_i) \} \]

With enumeration and union types mutually recursive data types can be built.

\[ e : \{c_1, c_2, \ldots, c_n\} \]

\[ \tau(e) = \{c_1, c_2, \ldots, c_n\} \]

\[ u : T_1 + T_2 + \ldots + T_n. \]

\[ \tau(u) = \tau(T_1) \cup \tau(T_2) \cup \ldots \cup \tau(T_n). \]
A logic program has a dual nature: It is a constraint system and a declaratively specified executable procedure.

\[(\text{Query}) \quad \text{query-name :? pattern-expression.}\]
\[(\text{Rule}) \quad f(t_1, t_2, \ldots, t_n) := \text{pattern-expression.}\]

A pattern introduces typed variables that range over the contents of an input model, and places constraints on those variables.

\[(\text{Pos}) \quad x \text{ is } f\]
\[(\text{Neg}) \quad \text{fail } x \text{ is } f\]
\[(\text{Constr}) \quad x_1 \{ = \mid \not= \} x_2\]
\[(\text{Arith}) \quad c \{ = \mid \not= \mid \leq \mid \geq \} \sum_i \prod_j x_{i,j}\]
Composition of Domains (I)

Operators for creating related domains

- Renaming operator
  - D as X

- Pseudo-(co)product
  - D1 */+ D2

- Type system
  - Type system 1
  - Type system 2
  - Safe Type system

- Logic Program
  - Logic Program 1
  - Logic Program 2
  - Rewritten Logic Program

- Conformance
  - Conformance 1
  - Conformance 2
  - Rewritten Conformance

- Type system 1 + 2
  - Type system 1
  - Type system 2

- Logic Program 1 + 2
  - Logic Program 1
  - Logic Program 2

- Conformance 1 + 2
  - Conformance 1
  - Conformance 2

\[ \text{models}(D \times [a, b]) \cong \text{models}(D_a) \times \text{models}(D_b). \]
Composition of Domains (II)

Operators for importing one domain into another.

\[ D_a \geq D_b \text{ if: } \exists D', D' \cong D_a \land \text{models}(D') \supseteq \text{models}(D_b). \]

There is also an “includes” operator that does not invoke any static analysis, and a “cextends” operator with is an application of “extends” followed by restricts.
Just Compose...

The total constraint system is constructed by composing and extending domains.

```
domain Architecture
  cextends (Functionality * Distribution)
{
  [function] Binding : (t: Task, d: Device).
  conflict :: Binding(t1,d), Binding(t2,d),
              Conflict(t1,t2).
  conforms :: !conflict.
}.
```

Note: There are a lot of constraints hiding in here!
A Design Space is a Set of Models

The models satisfying constraints form a design space.

domain Explore restricts (Architecture) {
    [bounds]
    candidates ??
    
    ////////// There are three different tasks
    t1 is Task, t2 is Task, t3 is Task,
    ////////// There are three different devices
    d1 is Device, d2 is Device, d3 is Device,
    ////////// There are three different channels
    ch1 is Channel, ch2 is Channel.
}

We use one more composition step to place bounds on the number of components/ECUs in the design space.
Solving Design Spaces
Using Symbolic Execution and Theorem Proving
Overview of Exploration Framework

1. **Formula Specification**
   - Symbolic Execution
   - Cardinality bounds on record instances

2. **SMT Formula**
   - Add symmetry breaking
   - Encode solution region

3. **Z3 Solver**
   - Reconstruct FORMULA model
   - Pick next region

4. **Try something new**
The input to symbolic execution is a FORMULA specification and a set of symbolic records.

\[
S = \{ 
    \text{Task}(x_1), \text{Task}(x_2), \text{Task}(x_3), \\
    \text{Device}(x_4), \text{Device}(x_5), \\
    \text{Conflict}(x_6, x_7), \text{Channel}(x_8, x_9, x_{10}), \\
    \text{Binding}(x_{11}, x_{12}) 
\} 
\]

Bounds can be inferred from static analysis. If not all records types can be bounded, then bounds are repeatedly guessed and tried.
Symbolic Execution

Symbolic execution produces a formula whose solutions (models) are in one-to-one correspondence with FORMULA models (sets of records).

\[ \text{test}_{Device}(d) \land \text{test}_{Channel}(in1) \land \text{test}_{Channel}(in2) \land \text{sel}_{Channel,1}(in1) = d \land \text{sel}_{Channel,1}(in2) = d \land in1 \neq in2 \land x = 2 \text{Int}(\text{sel}_{Channel,2}(in1)) + 2 \text{Int}(\text{sel}_{Channel,2}(in2)) \ldots \]

The resulting formula uses several theories.

**Term algebras**
- Recursive data types: Testers – to test if a record has a type
- Selectors – to select a field from a record

**Arithmetic**
- Typical arithmetic operations

**Quantifier-free**
- Quantifier-free formulas over uninterpreted functions with equality.
There are typically an infinite number of highly similar models. At the least we want to avoid non-isomorphic ones.

Generalization: Two FORMULA models are different if there does not exist a term-isomorphism between them.
Theorem provers are really good at finding the next solution:

However, this results in clustering of around solutions that are “almost” isomorphic.
Need to Direct Theorem Prover

Force the theorem prover to consider a region of the solution space.

Randomly construct a symbolic set of records, allow the solver to try all homomorphic images that are non-isomorphic to any attempt.
ExploreII($G, \varphi, \Pi$)
1: $solutions := \{\}$
2: $valid := \{\}$
3: $blocked := \{\}$
4: while True do
5:     $s := SampleClass(G, \Pi)$
6:     for all $p$ in $blocked$ do
7:         if $TestHomomorphism(p, s)$ then
8:             goto Line 4
9:         end if
10:    end for
11:    $C := \{\}$
12:    for all $q$ in $valid$ do
13:        $C := C \cup ComputeHomorphism(s, q)$
14:    end for
15:    $soln := FindModel(s \land \neg C \land \varphi)$
16:    if $soln \neq NULL$ then
17:        $valid := valid \cup \{Simplify(s, soln)\}$
18:        $solutions := solutions \cup \{soln\}$
19:    else
20:        if $CheckMostGeneral(s)$ then
21:            return $solutions$
22:        end if
23:        $blocked := blocked \cup \{s\}$
24:    end if
25: end while
Conclusion

FORMULA provides a novel framework for modeling and generalizing design space exploration.

The next step is to integrate DSE with rapid prototyping to explore global dynamics of constructed solutions.

FORMULA also supports model transformation and composition operators for transformations.