# Specifying and Composing Non-Functional Requirements in Model-based Development

Ethan K. Jackson[1], Dirk Seifert[2] and Markus Dahlweid[2]
Thomas Santen[2], Nikolaj Bjørner[1], and Wolfram Schulte[1]

[1]Microsoft Research,
One Microsoft Way, Redmond, WA
[2]European Microsoft Innovation Center,
Aachen, Germany
{ejackson,dseifert,mdahlwei,tsanten,nbjorner,schulte}@microsoft.com

**Abstract.** Non-functional requirements encompass important design concerns such as schedulability, security, and communication constraints. In model-based development they non-locally impact admissible platform-mappings and design spaces. In this paper we present a novel and formal approach for specifying non-functional requirements as constraint-systems over the space of models. Our approach, based on structured logic programming, allows interacting requirements to be specified independently from each other and composed together. Correct-by- construction operators eliminate some composition mistakes. Our approach is implemented in our formal modeling tool *FORMULA*, which can analyze the impacts of interacting non-functional requirements on platform mappings and design spaces.

## 1 Introduction

Model-based development (MBD) is a specification-driven approach to engineering complex software systems. It relies on design artifacts called *models*, which are specifications interpreted against a fixed and formalized context. MBD requires the precise description of the modeling context before any models are built. A canonical example from embedded systems demonstrates this idea:

Component $X$ produces an output event every 10 ms.

This specification is dangerous, because the concepts of component, event, and time are not elaborated. In order to apply MBD, the engineer must pick a context that gives precision to these concepts. These contexts go by many different names including: *domain* [1], *platform* [2], *modeling language* [3], and *model of communication and computation*(MoCC) [4]. We arbitrarily use the term *domain* in this paper. Domains are essential to the modeling process, because they provide the key assumptions about computation, communication, and time progression that enable pre-implementation analysis of software models.

There is a downside to the MBD approach. Engineers must occasionally define their own domains, resulting in the following complications:

– The engineer may not possess the skill-sets to formalize domain assumptions. For example, assumptions about time and concurrency are notoriously difficult to specify [5].
– Tools may not exist that are capable of analyzing models in the domain.
– System requirements may change, necessitating changes to the domain, its models, and associated tool-flow [6].

We shall refer to this overarching problem as the *domain specification problem*. The domain specification problem has been recognized, and motivated new research into libraries of composable domains [2, 4, 7, 8]. The hope is that whenever a new domain must be defined, it can be assembled from existing library elements so that existing tool-flows are utilized.

In this paper we examine the domain specification problem for *non-functional requirements*, which encompass important design concerns such as schedulability, security, and communication constraints. We show that many forms of non-functional requirements can be regarded as *constraints* over the space of models. In these cases domain specification reduces to formalizing a class of constraint-systems. Based on this observation, we have developed a novel methodology for integrating non-functional requirements into the MBD process:

– Engineers formalize non-functional requirements using algebraic data types and logic programming (LP). Logic programs have a precise execution semantics, so domains can be understood programmatically. However, our logic programs also have a precise interpretation as a system of first-order formulas, making them analyzable by existing tools. (Section 3)
– Our domains are composable via a novel set of operators acting on data types and logic programs. Our operators are *constructive*; they make semantic guarantees about composite domains. This property is crucial if composition techniques are to be an effective means for domain specification. (Section 4)
– We provide a design pattern, called the *Extend-Constrain-Merge Pattern*, by which engineers can separately specify and then compose non-functional requirements. Our correct-by-construction operators provide formal guarantees for this pattern. (Section 5)

We have implemented this work in our formal modeling tool *FORMULA* [9].

## 2 Background and Related Work

The domain specification problem has been recognized and studied from many angles; we summarize a few. Much research has been devoted to a common mathematical framework for the (straightforward) specification and composition of *behavioral semantics*. The *semantic anchoring* work of [7] uses *abstract state machines* (ASMs) as the mathematical underpinning for domains. The authors (and many others) argue that ASM-based formal methods are readily used by engineers. The *Ptolemy II* system [4] uses *interface automata* combined with a hierarchical composition scheme to formalize MoCCs and composition. Static

analysis is used to decide if the composite system is semantically meaningful. Recently, the *BIP* framework [8] uses a Plotkin-style *structural-operation semantics* (SOS). The domain specification problem has also been studied through denotational lenses. Recently, the *tagged-signal* formalism [10], based on trace algebras, reduces the problem to unifying various types of timelines. (We use the term "timeline" in a general sense.)

Behavioral semantics are useful for simulation and verification (when verification is possible), but there are other important modeling tasks. *Incremental refinement* is supported in MBD by automated translation of abstract specifications (platform-independent models) into implementation-dependent specifications (platform-dependent models) [11]. This process requires domains that express the rules by which abstract functionalities are mapped onto implementation components [12]. *Metamodeling* is a language-inspired approach for capturing domain rules. However, many have argued that existing metamodeling standards also lack formalization and composition operators [13]. *Design-space exploration* (DE) uses these rules as constraints to evaluate possible refinement paths [14]. Recent work in *feature diagrams* also takes a similar view on *software product lines* [15].

In this paper we focus on specifying and composing non-functional requirements for use in incremental refinement and design space exploration tasks. We avoid metamodeling standards to express domains, and instead develop a formal specification language using logic programming. Formally, our logic programs reduce to constraint-systems definable by first-order formulas over term algebras with restricted arithmetic constraints [16]. Domain composition is viewed as a composition of constraint-systems; we provide novel composition operators that make semantic guarantees about the composite domains. Automated theorem proving is supported by reduction to the state-of-the-art SMT (SAT modulo theories) solver *Z3* [17]. Thus, we make a trade-off between expressiveness and analyzability, which is similar to trade-off made by *Alloy* [18]. Our techniques are implemented in the tool *FORMULA* [9]. For the remainder of this paper we show how these techniques can be applied to non-functional requirements.

## 3 Non-functional Requirements As Constraint Systems

A common method of incremental refinement in MBD, called *platform mapping*, relies on the following design artifacts [12, 19]:

1. A *platform domain* (P domain) characterizing the space of implementation architectures at some level of abstraction.
2. A *platform-independent domain* (PI domain) characterizing the space of implementation-independent architectures at some level of abstraction.
3. A *platform model* (P model) of the target implementation architecture, which is a member of the platform domain.
4. A *platform-independent model* (PI model or just PIM) of the desired software architecture, which is a member of the PI domain.

The platform and platform-independent domains provide the necessary formal contexts for the two models. A platform mapping assigns elements from the PIM onto the platform model, thereby witnessing how the PIM is implemented.

*Example 1.* We now present a concrete example motivated by crucial abstractions found in safety-critical systems [20, 21]. A platform model contains a (heterogeneous) set of processors connected by point-to-point communication buses, for which the maximum communication delay is known. A set of known software functions can be executed on a processor with known *worst-case execution times* (WCETs). (For simplicity, we assume WCETs can be summed.) Intuitively, the platform domain is the set of all such models.

A platform-independent model consists of a set of software components connected together by communication channels. Each software component utilizes a set of functions. Component $c_j$ executes all of its functions $n_j$ times every $T$ units of time. The platform-independent domain is the set of all such models. A platform mapping assigns each component to a processor so that on processor $p_i$:

$$\sum_{c_j \in p_i} \left( n_j \sum_{f_k \in c_j} wcet(f_k, p_i) \right) \leq T \tag{1}$$

where $c_j$ is a component mapped onto $p_i$. Also, $f_k$ is a function used by $c_j$ and $wcet(f_k, p_i)$ is the WCET of function $f_k$ on $p_i$. (We ignore communication jitter and connectivity for the moment.) □

Equation 1 embodies an important non-functional requirement: *schedulability*. This requirement affects incremental refinement by reducing the total number of legal platform mappings from the PI domain to the P domain. Turning this around, we can define a composite domain $D_{mapping}$ whose models can be decomposed into a platform model, a component model, and a function from components to processors satisfying Equation 1. If a model belongs to the $D_{mapping}$ domain, then it is a valid architecture satisfying the schedulability requirement. Conversely, enumerating models from $D_{mapping}$ yields a subset of valid refinements.

This example illustrates that many non-functional requirements can be regarded as constraint-systems. However, three enabling technologies are required to apply this observation for modeling: (1) A technology for specifying domains as constraint-systems. (2) Composition operators that guarantee some relationships between conforming models. (3) Modeling finding procedures that enumerate domain models.

### 3.1 Specifying Domains

The previous example shows that domains contain data structures for representing models. We use *free constructors* as a simple, yet flexible, formalization of data structures. An $n$-ary free constructor is a partial function $f$ that takes $n$

```
1  /// Platform-independent domain
2  domain Component {
3    /// Data structures
4    [multiplicity(1,*)]
5    Cmp : (id: Basic).
6    [relation]
7    Cnn : (from: Cmp, to: Cmp).
8    [multiplicity(1,1)]
9    Period : (time: PosInteger).
10   [function]
11   Freq : (cmp: Cmp, freq: PosInteger).
12   [relation(cmp)]
13   Uses : (cmp: Cmp, func: String).
     /// Some additional constraints
15   bad_Cnn1 :? cn is Cnn, fail cm is Cmp,
16               cn.from.id = cm.id.
17   bad_Cnn2 :? cn is Cnn, fail cm is Cmp,
18               cn.to.id = cm.id.
19   atleast :? cm is Cmp.
     /// conforms query
21   conforms :? atleast & !bad_Cnn1 &
22               !bad_Cnn2.
23 }
```

```
25 /// Platform domain
26 domain Platform {
27   [multiplicity(1,*)]
28   Prc : (id: Basic).
29   [relation(from,to)]
30   Cnn : (from: Prc, to: Prc,
31          delay: PosInteger).
32   [multiplicity(1,*)]
33   Func : (id: String).
34   [cartesian(from,to)]
35   Wcet : (func: Func, proc: Prc,
36           wcet : PosInteger).
   }
```

**Fig. 1.** Specifying the PI and P domains with FORMULA.

arguments $t_1, t_2, \ldots, t_n$ and returns a new data instance $t$ if all arguments are correctly-typed: $t = f(t_1, t_2, \ldots, t_n)$. Two instances of data $t$ and $t'$ are equal if and only if: (1) Both $t$ and $t'$ are the same constant (e.g. $t = t' = 1$), (2) Or $t$ and $t'$ were constructed by the same free constructor $f$ applied to the same list of arguments. In our framework a *model* is a finite set of data instances constructed by free constructors. A model $M$ is a member of a domain $D$ if its data elements were built from constructors of $D$ and $M$ satisfies the constraints of $D$.

Lines 2-23 in Figure 1 contain the specification for the platform-independent domain written in our FORMULA language. Line 2 declares the PI domain with the name Component. Line 5 declares a free constructor called Cmp, which is used to construct instances of components. The single argument to Cmp is an identifier of type Basic, i.e. the argument can be a string or numeric constant. Line 7 introduces another constructor Cnn for constructing connections between components. In this case, both arguments to Cnn must be instances of Cmp. The other constructors are:

- Period for declaring the amount of time $T$ it takes for all components to complete all their computations.
- Frequency for assigning the number of times $n_j$ a particular component must repeat its computation within $T$ units of time.
- Uses gives the String names of software functions used by component $c_j$.

A model of the Component domain is a set of data instances built using these constructors, which also satisfies the domain constraints. For example:

$$M = \left\{ \begin{array}{c} Period(10), \quad Cmp(1), \quad Cmp(2), \\ Cnn(Cmp(1), Cmp(2)), \\ Freq(Cmp(1), 1), \quad Freq(Cmp(2), 2) \\ Uses(Cmp(1), \text{``TempSensor''}), \quad Uses(Cmp(2), \text{``FFT''}) \end{array} \right\}. \quad (2)$$

This model has two components that complete computations every 10 time units. Component 1 executes the TempSensor function once every 10 units and connects to component 2. Component 2 uses the FFT function twice every 10 units.

**Logic Programming.** Domain constraints are specified with *logic programming*. In the interest of space, we provide an informal discussion of logic programs. Please see [16] for a formal presentation. Intuitively, our logic programs are pattern matching (i.e. *unification*) procedures that search the data contained by models. The results of these searches are used to decide if a model belongs to the domain. A logic program has just two forms of statements: *queries* and *rules*.

(Query) query-name :? pattern-expression.
(Rule) f($t_1$,$t_2$,...,$t_n$) :- pattern-expression.

A query is effectively a boolean variable whose value is true if the pattern is matched at least once; otherwise it takes the value false. A rule instantiates a new data element using free constructor $f$ every time its pattern is satisfied. This new data does not change the input model, but it can be observed by other queries/rules allowing interactions between statements of the program. Pattern expressions are formed from four basic constructs:

(Pos) x **is** f (Neg) **fail** x **is** f
(Constr) $x_1$ { = | ≠ } $x_2$ (Arith) c {= | ≠ | ≤ | ≥} $\sum_i \prod_j x_{i,j}$

A positive (Pos) expression searches the model for a data instance $t$ of type $f$; if it succeeds then $x$ is bound to $t$: $x = t$. A variable appearing in a positive expression is called a *positive variable*. A constraint (Constr) expression tests if two data instances are equal; $x_{i \in \{1,2\}}$ must be a variable or a data instance. An arithmetic (Arith) expression compares sums-of-products to a numeric constant $c$; every $x_{i,j}$ must either be a constant or variable capable of taking a numeric value. A negative (Neg) expression checks that there is no data instance of type $f$ satisfying all the constraints on $x$. Negative expressions do not create bindings.

A pattern expression is a list of basic constructs, and is satisfied for all positive variable bindings that cause every construct to be satisfied. For example, lines 15-16 in Figure 1 search for connections (Cnn) with missing starting-points:

bad_Cnn1 :? cn **is** Cnn, **fail** cm **is** Cmp, cn.from.id = cm.id.

This query is satisfied if the variable cn can be bound to a connection such that the starting-point of the connection does not exist. It is equivalent to the following first-order statement:

$$bad\_Cnn1 \Leftrightarrow \begin{array}{ll} \exists x, y, z & x = Cmp(z) \wedge Cnn(x, y) \in M \wedge \\ (\neg \exists w & Cmp(w) \in M \wedge w = z). \end{array} \tag{3}$$

Similarly, lines 17-18 find connections with missing end-point. Line 19 checks that at least one component exists in the model.

**Model Conformance.** Every domain provides a special query called conforms for characterizing the models belonging to that domain. A model $M$ belongs to a domain $D$ if the conforms query of $D$ evaluates to true for $M$. The conforms query is unique to our approach and enables composition of domains. It can be defined with a pattern expression, or, for convenience, it may be a boolean combination of existing queries. For example:

**conforms** :? atleast & !bad_Cnn1 & !bad_Cnn2.

accepts any model with at least one component (line 19), and where the bad_Cnn1 and bad_Cnn2 queries evaluate to false.

A typical domain will introduce many queries which validate the correctness of model data. In order to reduce the specification burden, the free constructors can be annotated to indicate common usage patterns. For example, line 6 annotates the Cnn constructor with the [relation] annotation. This causes the implicit introduction of the queries in lines 15-18 and implicitly conjuncts their negations onto the conforms query. Here are several common usage patterns that can be directly translated into logic programming: (Each pattern has two forms.)

- [relation], [relation($c_1$,$c_2$,...,$c_m$)]: (1) Every argument of every $f$ data-instance must exist in $M$. (2) Every argument of the $c_i$ component of every $f$ data-instance must exist in $M$. (The set of numeric and string constants exist by default.)
- [cartesian], [cartesian($c_1$,$c_2$,...,$c_m$)]: (1) The constructor $f$ is an $n$-ary relation, and if $t_1, \ldots, t_n$ are in $M$ and $f$ can be applied to this list, then there is an instance $t = f(t_1, \ldots, t_n)$ in $M$. (2) The projection of all $f$ instances onto the components ($c_1$,$c_2$,...,$c_m$) is cartesian.
- [function], [function($c_1$,$c_2$)]: (1) The constructor $f$ is a binary relation and the projection of $f$ onto its first component is cartesian. (2) $f$ is an $n_{\geq 2}$-ary constructor and the projection of $f$ onto components ($c_1$,$c_2$) is a function.
- [multiplicity($k_{min}$, $k_{max}$)], [multiplicity($k_{min}$, *)]: (1) There are between $k_{min}$ and $k_{max}$ (inclusive) unique instances of $f$ in $M$. (2) There are at least $k_{min}$ unique instances of $f$ in $M$.

Lines 25-36 of Figure 1 characterize the platform domain using annotations to implicitly define many queries, including the conforms query. Briefly, Prc constructs processors in the platform. Cnn connects processors and specifies the

maximum communication delay. Func declares software functions that execute on processors. Wcet defines the worst-case execution time of a function $f$ on processor $p$. In summary, we have shown that platform-independent and platform domains can be specified using algebraic data types, logic programming, and the conforms query. This provides a declarative pattern-matching style for formalizing domains as constraint-systems over first-order logic (with the theories of term algebras and arithmetic).

## 4 Composing Domains

Next, we wish to specify $D_{mapping}$, which formalizes a non-functional requirement in terms of the PI and P domains. Following the previous example, it is tempting to declare a composite domain $D_{mapping}$ that lexically imports the Component and Platform domains. In fact, lexical importation is the primary means of composition in *UML*, a standard language for MBD [19]. However, this approach may introduce errors since lexical importation makes no semantic guarantees about the resulting constraint systems.

This problem already appears in our simple example: Combining the two specifications leads to a conflict because both specifications declare the Cnn constructor, but with different types (lines 7 and 29). Actually, such a conflict is the *best-case* scenario, because it draws attention to the problem. Consider the alternative where both domains compatibly define Cnn. Then, the two logic programs can non-locally interact through shared data. This interaction may yield an unexpected constraint system that does not represent the intended composition. Non-local interaction is not just a property of our framework, but arises whenever specifications can arbitrarily share concepts. Again, UML exhibits this same problem [13].

**The Product Construction.** In our approach we do not eliminate non-local interactions, but provide correct-by-construction operators for controlling interactions. First, we introduce some terminology. Let $models(D)$ denote all finite sets of data instances satisfying the conforms query of $D$. Two sets of models $\mathbf{M}_1$ and $\mathbf{M}_2$ are *isomorphic*, written $\mathbf{M}_1 \cong \mathbf{M}_2$, if there exists a bijection $\varphi : \mathbf{M}_1 \to \mathbf{M}_2$. We extend isomorphism to domains: $D_a \cong D_b$ if $models(D_a) \cong models(D_b)$. The first step towards controlling interaction is the construction of the *product domain* $D_\times[a, b]$, which has the property:

$$models(D_\times[a, b]) \cong models(D_a) \times models(D_b). \tag{4}$$

When context is clear, we write $D_\times$ for the product of $D_a$ and $D_b$. The constraint system $D_\times$ eliminates all interactions between domains, which serves as an ideal starting point for controlled composition. Another way to understand the product is through projections $\pi_a : models(D_\times) \to models(D_a)$ and

$\pi_b : models(D_\times) \to models(D_b)$. Then, the models satisfy:

$$\Big(M_\times \in models(D_\times)\Big) \Leftrightarrow \Big(\pi_a(M_\times) \in models(D_a) \wedge \pi_b(M_\times) \in models(D_b)\Big) \tag{5}$$

We provide two operators for constructing the product domain: The *pseudo-product* (denoted '$*$') and the *renaming operator* (denoted '$\mathsf{as}$'). The pseudo-product lexically imports all definitions from $D_a$ and $D_b$ into a composite domain named $(D_a * D_b)$. Inside the composite domain each query $q$ of $D_a$ is renamed to $D_a.q$ and each query $q'$ in $D_b$ is renamed to $D_b.q'$. Then, the $\mathsf{conforms}$ query of the composite becomes:

$$\mathsf{conforms} :? \ \mathsf{D}_a.\mathsf{conforms} \ \&\ \mathsf{D}_b.\mathsf{conforms}.$$

This construction is precisely the product domain if a certain condition holds. Given a free constructor $f$ in domain $D$, let $[\![f]\!]$ be the set of all data instances that can be constructed by $f$, then:

$$\mathcal{T}(D) = \bigcup_{f \in D} [\![f]\!]. \tag{6}$$

**Lemma 1.** *Given domains $D_a$ and $D_b$ such that $\mathcal{T}(D_a) \cap \mathcal{T}(D_b) = \emptyset$ then:*

$$D_\times[a, b] \cong (D_a * D_b). \tag{7}$$

**Proof Sketch.** Define projection operators:

$$\pi_a(M) \mapsto \Big(M \cap \mathcal{T}(D_a)\Big), \quad \pi_b(M) \mapsto \Big(M \cap \mathcal{T}(D_b)\Big). \tag{8}$$

Verify that projections satisfy Equation 5. □

This observation explains that if two specifications are disjoint, then their product can be easily formed. However, in our example the specifications are not disjoint (due to $\mathsf{Cnn}$), so ($\mathsf{Component}$ * $\mathsf{Platform}$) is not the product. In this case, the renaming operator '$\mathsf{as}$' is to used establish disjointness. Let $D' = D$ $\mathsf{as}$ $X$, then $D'$ contains all definitions of $D$, except that every declaration/use of a function symbol $f$ is replaced with the declaration/use $X.f$. Every query $q$ is renamed to $X.q$, and the $\mathsf{conforms}$ query of $D'$ is $\mathsf{conforms}$ :? $\mathsf{X.conforms}$.

**Lemma 2.** *Given domains $D_a$ and $D_b$. Let $D' = D_a$ $\mathsf{as}$ $X$ and $D'' = D_b$ $\mathsf{as}$ $Y$ for $X \neq Y$. Then, $\mathcal{T}(D') \cap \mathcal{T}(D'') = \emptyset$.*

**Proof Sketch.** Assume there exists an element $t \in \mathcal{T}(D') \cap \mathcal{T}(D'')$, then there must exist free constructors $f' \in D'$ and $f'' \in D''$ such that:

$$f'(t_1, t_2, \ldots, t_n) = t = f''(s_1, s_2, \ldots, s_m). \tag{9}$$

However, since $f'$ and $f''$ are free constructors, it must be that $f'$ and $f''$ are the same free constructor in order for Equation (9) to hold. This contradicts that $D'$ and $D''$ have disjoint free constructors due to the renaming operations: $D' = D_a$ $\mathsf{as}$ $X$ and $D'' = D_b$ $\mathsf{as}$ $Y$. □

**Theorem 1.** *Let $(D_i)_{i \in I}$ be a finite collection of domains indexed by $I$. Let $rename : I \to \mathcal{V}$ be an injective function onto some vocabulary of names $\mathcal{V}$. Then:*

$$(D_{i_1} \textsf{ as } rename(i_1)) * (D_{i_2} \textsf{ as } rename(i_2)) * \ldots * (D_{i_n} \textsf{ as } rename(i_n)) \cong D_{\times}[i_1, i_2, \ldots, i_n]. \tag{10}$$

**Proof Sketch.** The $*$ operator is associative, because lexical importation and boolean conjunction are associative. (The order in which domain definitions are imported is semantically irrelevant in FORMULA.) Therefore, by pairwise disjointness of free constructors and repeated application of lemmas (1) and (2) the overall product is constructed. □

Thus, our constructive composition operators provide semantics guarantees about the composite constraint system.

**Extending and Restricting.** Returning to our example, we can safely construct the product domain (Component as C * Platform as P). However, this system still lacks the platform-mapping machinery. The next problem is to extend the product with new data without disturbing existing properties. We call domain $D_a$ an *extension* of $D_b$, written $D_a \geq D_b$ if:

$$\exists D',\ D' \cong D_a \wedge models(D') \supseteq models(D_b). \tag{11}$$

Conversely, $D_b$ is a *restriction* of $D_a$. These properties formalize set-theoretic containment up to isomorphism. Again, we provide the operators *extends* and *restricts* for constructively building domains with property 11. Briefly, both operators lexically import definitions and rename queries. The conforms query of an extended domain is the disjunction of all locally defined conformances with imported conformances. On the other hand, a restricted domain is a conjunction of all local conformances with imported conformances; restricted domains cannot introduce new free constructors. Finally, syntactic analysis of constituent logic programs ensures that logic programs do not communicate information that non-locally effects conformance. (This check only uses unification, so it is efficient.)

**Theorem 2.** *Let ($D_a$ extends $D_b$) be a domain $D_a$ extending $D_b$, which satisfies all syntactic checks of the extends operator. Then $D_a \geq D_b$. Let ($D_a$ restricts $D_b$) be a domain $D_a$ restricting $D_b$, which satisfies all syntactic checks of the restricts operator. Then $D_a \leq D_b$.*

Figure 2 shows the final FORMULA specification for the schedulability requirement. Lines 37-42 specify the Mapping domain, which extends the product (Component as C * Platform as P) with a platform-mapping function from components (C.cmp) to processors (P.prc). Notice that neither Cmp nor Prc are qualified with the renaming prefixes "C." and "P." (Line 40). In this case qualification is unnecessary, because FORMULA can deduce the prefixes. (This is not the case for

```
37. domain Mapping extends (Component as C * Platform as P) {
      /// Adds mapping to product
39.   [function]
40.   Map    : (cmp: Cmp, proc: Prc).
41. }
42. domain Scheduling restricts (Mapping) {
      /// Derived constructor not used in models
44.   cwcet : (cmp: Cmp, wcet: Natural).

46.   cwcet(c, 0) :- c is Cmp, fail u is Uses, u.cmp = c.
47.   cwcet(c, t) :- c is Cmp,
48.                  bagof{z | u is Uses, m is Map,
49.                            f is Freq, w is Wcet,
50.                            u.cmp = c, u.func = w.func.id,
51.                            m.cmp = c, m.proc = w.proc,
52.                            f.cmp = c,
53.                            z = w.wcet*f.freq
54.                       }(times), sum{tp | times}(t).

56.   unsched(p)  :- p is Prc, per is Period,
57.                  bagof{w.wcet | m is Map, w is cwcet,
58.                                 m.proc = p, m.cmp = w.cmp,
59.                       }(wcets), sum{tp | wcets}(t), t > per.time.
60.   isMapped :? m is Map.
61.   isSched  :? fail un is unsched.
62.   conforms :? isMapped & isSched.
63. }
```

**Fig. 2.** Extending and restricting domains. (Con't from 1.)

Cnn.) This feature helps to minimize the impact of renaming on the complexity of the specification.

The Scheduling domain restricts Mapping by eliminating unschedulable architectures. Line 44 introduces a *derived constructor* cwcet, which only stores temporary information during execution of the logic program. Derived constructors always start with a lowercase letter and never appear in models, so they can be added to restricted domains. (Also, derived constructors need not be explicitly declared.) The rules at lines 46 and 47-54 calculate the total WCET for components and store this as cwcet data. Line 46 deduces that components using no functions have WCETs of zero. The other rule employs two special constructs (common in logic programming) bagof and sum to find the total WCET of components using one or more functions:

**bagof** { var | pattern-expression }(derived-constructor)

The bagof operator finds all matches to pattern-expression. A data instance derived-constructor($s$, $n$) is constructed for each unique binding var = $s$. The

second argument $n$ is the number of times that var took the binding $s$ over all matches. In this rule the pattern expression determines how the component c has been mapped onto the platform, and then adds w.wcet*f.freq to the bag for each function u.func used by c. In other words, a function that executes f.freq times per interval and takes w.wcet time per execution contributes w.wcet*f.freq time units to the WCET of the component. Next, the sum operator sums the values in a bag (multiset) and binds the result to a variable. Thus, $t$ (line 54) is the total WCET of component c.

Finally, lines 56-59 sum the worst-case executions of all components assigned to processor p. If this sum is greater than the specified period of the system, then the system is unschedulable. The conforms query requires no unschedulable processors and requires the platform-mappings to be present. Implicitly, the conforms query is conjuncted with Mapping.conforms which, in turn, has C.conforms and P.conforms as conjuncts. In conclusion, the schedulability requirement has been formalized in a compact step-wise fashion with semantic guarantees at each step. Using techniques described in [9], the platform-mapping problem can be automatically solved and valid mappings can be enumerated. For our schedulability example, these tasks amount to solving an NP-hard problem [20], further illustrating the importance of properly formalizing non-functional requirements.

## 5 Composing Many Non-functional Requirements

Realistic software systems exhibit multiple competing non-functional requirements (*design concerns*). This complicates the job of the modeler, who must formalize each requirement and the interactions between requirements. For example, suppose we want a new *security* requirement:

*Example 2.* Every component has a security level, which is a non-negative integer. A platform-mapping assigns components to processors, such that all components on processor $p$ have the same security level. □

The security requirement interacts with schedulability in complex ways; it may be impossible to simultaneously satisfy security and schedulability in a particular architecture. This is an instance of a more general problem: Engineers need better mechanisms for separately specifying and then composing competing non-functional requirements [5]. We present such a design pattern, called the *Extend-Constrain-Merge Pattern*, for separate specification and then composition (Figure 3). We now describe the key steps of the pattern.

**Step 0: Foundations.** In the zero$^{th}$ step of the pattern, the modeler characterizes the basic notions of "component" and "platform". These serve as the foundations for later steps. The process begins by defining a *minimally-complex* abstraction level for components. We call this domain Component$_{min}$; it contains the *minimum* data/constraints needed to represent high-level architectures as sets of components. All later non-functional requirements must be reconciled

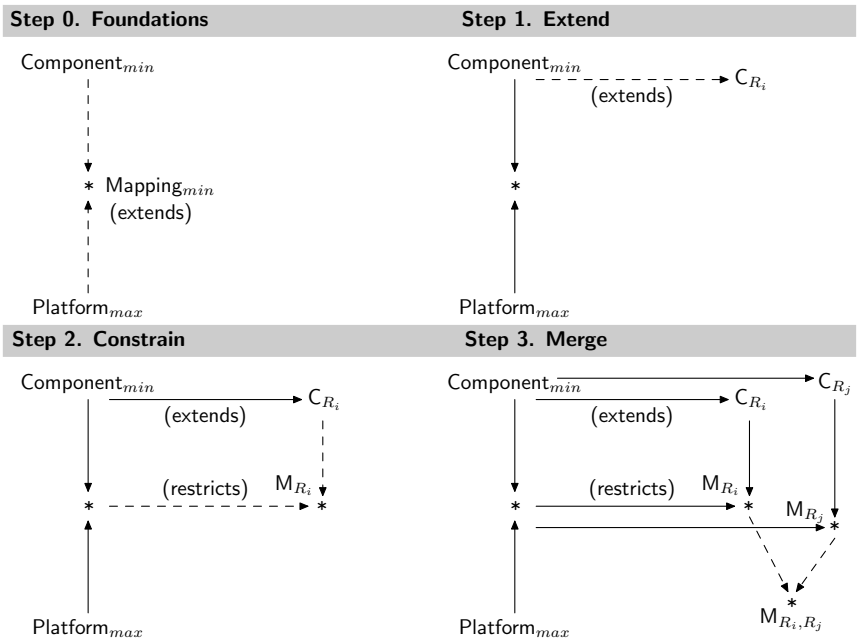



**Fig. 3.** Steps of the Extend-Constrain-Merge Pattern.

against these elements. Second, a *maximally-detailed* abstraction of the platform is defined. By maximally-detailed we mean as much detail as possible should be formalized while remaining at the desired abstraction level. (For example, the cycle-accurate behaviors of software functions are below our abstraction level, but WCETs are not.) Early maximization of platform detail provides a common language with which to formalize non-functional requirements as the need arises. Third, a domain $\mathsf{Mapping}_{min}$ extends the product of $\mathsf{Component}_{min}$ with $\mathsf{Platform}_{max}$:

$$\mathsf{Mapping}_{min} \text{ extends } (\mathsf{Component}_{min} \text{ as C * } \mathsf{Platform}_{max} \text{ as P}) \{\ldots\}.$$

$\mathsf{Mapping}_{min}$ captures the minimum data/constraints needed to construct platform-mappings given maximum platform detail. The remaining three steps are repeated on a per-requirement basis.

**Step 1: Extend.** To add a new requirement $R$, begin by defining a component domain $\mathsf{C}_R$ extending $\mathsf{Component}_{min}$:

$$\mathsf{C}_R \text{ extends } (\mathsf{Component}_{min}) \{\ldots\}.$$

$\mathsf{C}_R$ contains all additional data/constraints needed to describe this requirement (e.g. *"components have security levels"*). Encapsulating these elements in $\mathsf{C}_R$ isolates potential specification errors.

**Step: 2 Constrain.** Next, define a mapping domain $\mathsf{M}_R$ of the following form:

$$\mathsf{M}_R \text{ restricts } (\mathsf{C}_R \text{ as R * Mapping}_{min} \text{ as M}) \{\ldots\}.$$

Note that R should be a unique renaming prefix, and the renaming prefix M should be fixed for all $\mathsf{M}_R$. $\mathsf{M}_R$ *constrains* the legal platform-mappings in the following way: Given a model belonging to $\mathsf{C}_R$, then $\mathsf{M}_R$ should remove all mappings from $\mathsf{Mapping}_{min}$ that cannot implement the system described by the model. Since $\mathsf{Mapping}_{min}$ contains $\mathsf{Platform}_{max}$, $\mathsf{M}_R$ can examine all the details of the platform in order to make this decision. However, $\mathsf{M}_R$ does not have access to data used by other requirements, thereby enforcing a separation of concerns.

**Step 3: Merge.** The constructions in steps 1-2 guarantee that requirements are formalized separately. In this step separately defined components are composed together to formalize their interaction. The interactions between requirements $\mathsf{R}_1$, $\mathsf{R}_2$, ..., $\mathsf{R}_n$ are captured by their pseudo-product:

$$(\mathsf{M}_{R_1} * \mathsf{M}_{R_2} * \ldots * \mathsf{M}_{R_n}).$$

This pseudo-product represents the set of all platform mappings that simultaneously satisfy competing requirements. Formally, composition is understood with the following theorem:

**Theorem 3.** *Given a collection of separately specified requirements* $(\mathsf{C}_{R_i}, \mathsf{M}_{R_i})_{i\in I}$, *define a collection of projection operators.*

$$\pi_{R_i} : models(\mathsf{M}_{R_i}) \to models(\mathsf{Mapping}_{min}). \tag{12}$$

*and*

$$\pi_{interact} : models(\mathsf{M}_{R_1} * \mathsf{M}_{R_2} * \ldots * \mathsf{M}_{R_n}) \to models(\mathsf{Mapping}_{min}). \tag{13}$$

*then:*

$$\bigcap_{i\in I} \mathbf{img}\ \pi_{R_i} = \mathbf{img}\ \pi_{interact}. \tag{14}$$

In other words every legal model of the pseudo-product contains a platform-mapping such that all requirements agree on the correctness of this mapping. This is accomplished without any *a priori* mixing of requirements.

**Visualizing the Composition.** Figure 4 shows a visualization of Theorem 3. The rectangular region labeled $models(\mathsf{Mapping}_{min})$ contains all possible platform mappings from every possible component model to every possible platform model. Each non-functional requirement $R_i$ identifies some subset of components/platforms/mappings capable of satisfying the requirement. This subset is obtained by a projection function $\pi_{R_i}$ that takes every legal platform mapping under the extended language of $M_{R_i}$ and forgets these extensions, thereby returning a valid combination in the language of $\mathsf{Mapping}_{min}$. In order for some

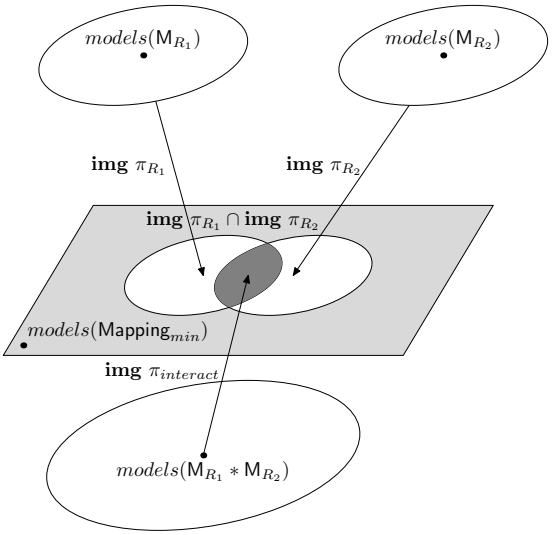


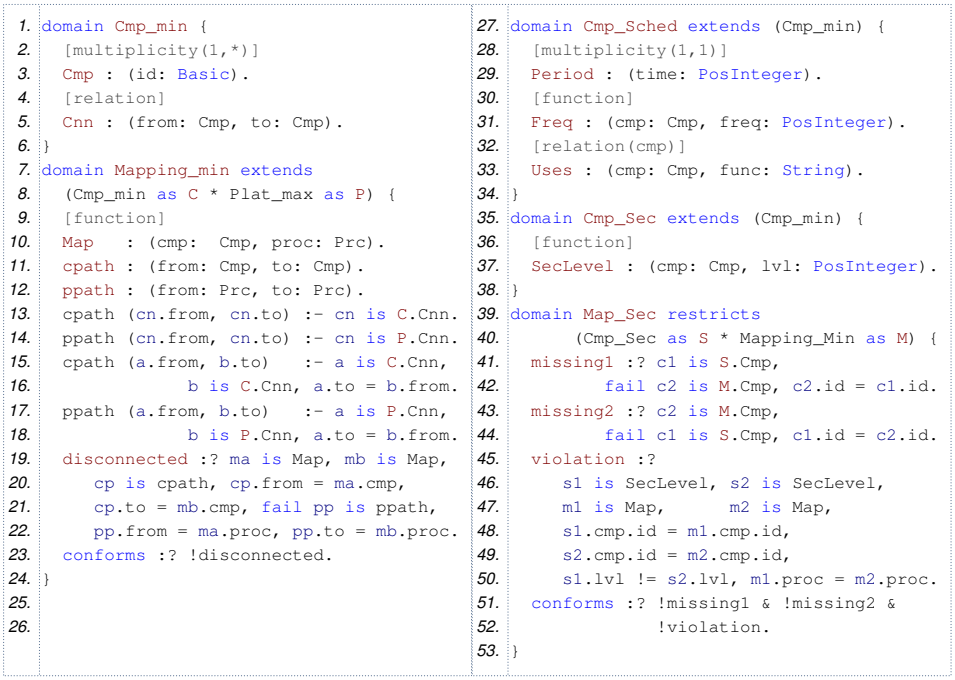

**Fig. 4.** Visualizing the composition of separately specified non-functional requirements.

architecture to simultaneously satisfy multiple non-functional requirements, it must be in the intersection of these projections. The figure illustrates the intersection of two such non-functional requirements by the darkly-shaded region.

However, finding the elements of this intersection from the specifications $M_{R_1}$ and $M_{R_2}$ is a non-trivial task. The extend-constrain-merge pattern solves this problem by constructing a new specification $(M_{R_1} * M_{R_2})$ where the projection onto $models(\mathsf{Mapping}_{min})$ is exactly the intersection of the independently specified non-functional requirements. This projection operator is labeled $\pi_{interact}$ in the figure. By using the right composition mechanisms it is possible to correctly compose and reason over constraint systems in order to find architectures that simultaneously satisfy many non-functional requirements.

**Applying The Pattern.** Figure 5 shows the extend-constrain-merge pattern applied to the previous examples. Assume $\mathsf{Platform}_{max}$ is the domain given in lines 25-36 of Figure 1. $\mathsf{Component}_{min}$ consists of only components and connections (lines 1-6). Essentially, it is just a set of finite, labeled, and directed graphs. All later non-functional requirements must use this notation for components. The $\mathsf{Mapping}_{min}$ domain (lines 7-24) formalizes the mapping problem given a component model with minimum detail and a platform model with maximum detail. $\mathsf{Mapping}_{min}$ discards mappings that do not respect connectivity of components. Lines 13-18 calculate the transitive closure[1] of a component model

[1] In logic programming transitive closure is an example of a *recursive program*. We do not allow true recursion in FORMULA, so the user must bound the depth of the recursion with respect to the size of the input model.



```
1.  domain Cmp_min {
2.    [multiplicity(1,*)]
3.    Cmp : (id: Basic).
4.    [relation]
5.    Cnn : (from: Cmp, to: Cmp).
6.  }
7.  domain Mapping_min extends
8.    (Cmp_min as C * Plat_max as P) {
9.    [function]
10.   Map   : (cmp:  Cmp, proc: Prc).
11.   cpath : (from: Cmp, to: Cmp).
12.   ppath : (from: Prc, to: Prc).
13.   cpath (cn.from, cn.to) :- cn is C.Cnn.
14.   ppath (cn.from, cn.to) :- cn is P.Cnn.
15.   cpath (a.from, b.to)   :- a is C.Cnn,
16.                b is C.Cnn, a.to = b.from.
17.   ppath (a.from, b.to)   :- a is P.Cnn,
18.                b is P.Cnn, a.to = b.from.
19.   disconnected :? ma is Map, mb is Map,
20.      cp is cpath, cp.from = ma.cmp,
21.      cp.to = mb.cmp, fail pp is ppath,
22.      pp.from = ma.proc, pp.to = mb.proc.
23.   conforms :? !disconnected.
24. }
25.
26.
27. domain Cmp_Sched extends (Cmp_min) {
28.   [multiplicity(1,1)]
29.   Period : (time: PosInteger).
30.   [function]
31.   Freq : (cmp: Cmp, freq: PosInteger).
32.   [relation(cmp)]
33.   Uses : (cmp: Cmp, func: String).
34. }
35. domain Cmp_Sec extends (Cmp_min) {
36.   [function]
37.   SecLevel : (cmp: Cmp, lvl: PosInteger).
38. }
39. domain Map_Sec restricts
40.       (Cmp_Sec as S * Mapping_Min as M) {
41.   missing1 :? c1 is S.Cmp,
42.          fail c2 is M.Cmp, c2.id = c1.id.
43.   missing2 :? c2 is M.Cmp,
44.          fail c1 is S.Cmp, c1.id = c2.id.
45.   violation :?
46.      s1 is SecLevel, s2 is SecLevel,
47.      m1 is Map,      m2 is Map,
48.      s1.cmp.id = m1.cmp.id,
49.      s2.cmp.id = m2.cmp.id,
50.      s1.lvl != s2.lvl, m1.proc = m2.proc.
51.   conforms :? !missing1 & !missing2 &
52.               !violation.
53. }
```

**Fig. 5.** Using the Extend-Constrain-Merge Pattern to specify multiple requirements.

and platform model. A mapping is correct if every pair of reachable components is mapped to a pair of reachable processors (lines 19-23).

Next, the scheduling requirement is refactored so that timing-specific constructs are isolated to the Cmp_Sched domain (lines 27-34). Meanwhile, the Cmp_Sec domain introduces the security level (SecLevel) concept (lines 35-38). Both domains extend $\mathsf{Component}_{min}$, thereby inheriting the components-as-graphs notation. Finally, the Map_Sec domain throws out systems that map components with different security levels to the same processor (lines 39-52). The Scheduling domain (Figure 2) can be similarly refactored into Map_Sched, though we omit this in the interest of space. Finally, platform-mappings satisfying both requirements are described by (Map_Sched * Map_Sec).

## 6 Conclusions and Future Work

In conclusion, we showed that non-functional requirements can be regarded as constraint systems over the space of platform-mappings. This space is an extension of the product of the platform-independent domain and platform domain. We described several correct-by-construction composition operators for building non-functional requirements out of smaller constraint systems. Finally, we presented the extend-constrain-merge pattern as a concrete method for applying our techniques.

The techniques are implemented in the FORMULA language, which uses logic programming as the basis for a specification language. FORMULA specifications are translated into state-of-the SMT solvers for evaluation and construction of platform-mappings. We believe that logic programming serves as a reasonable specification language for non-functional requirements, since it is an intermediary between fully operational program-like specifications and fully relational algebraic specifications.

Currently we are exploring the scalability of this technique by developing FORMULA specifications for realistic non-functional constraints. For example, we have developed component models and encoded schedulability requirements for the *Timing Definition Language* (TDL) [22], which is used for hard real-time embedded systems. We are also studying how security requirements expressed by high-level authorization languages, e.g. *DKAL* [23] or *SecPAL* [24], may be reflected as constraints systems on platform mappings. Finally, we are prototyping code synthesis tools that generate candidate implementations from the results of design-space exploration.

## References


1. Jackson, M., Zave, P.: Domain descriptions. Proceedings of IEEE International Symposium Requirements Engineering (RE) (Jan 1993) 56–64
2. Pinto, A., Bonivento, A., Sangiovanni-Vincentelli, A.L., Passerone, R., Sgroi, M.: System level design paradigms: Platform-based design and communication synthesis. ACM Trans. Design Autom. Electr. Syst. **11**(3) (2006) 537–563
3. Karsai, G., Sztipanovits, J., Lédeczi, Á., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (2003) 145–164
4. Lee, E.A., Xiong, Y.: A behavioral type system and its application in Ptolemy II. Formal Asp. Comput. **16**(3) (2004) 210–237
5. Henzinger, T.A., Sifakis, J.: The Embedded Systems Design Challenge. In: Proceedings of the International Symposium on Formal Methods (FM). (2006) 1–15
6. Sprinkle, J., Karsai, G.: A domain-specific visual language for domain model evolution. J. Vis. Lang. Comput. **15**(3-4) (2004) 291–307
7. Chen, K., Sztipanovits, J., Neema, S.: Compositional specification of behavioral semantics. In: Proceedings of Design, Automation and Test in Europe Conference (DATE). (2007) 906–911
8. Bliudze, S., Sifakis, J.: A Notion of Glue Expressiveness for Component-Based Systems. In: Proceedings of the International Conference on Concurrency Theory (CONCUR). (2008) 508–522
9. Jackson, E., Schulte, W., Sztipanovits, J.: The Power of Rich Syntax for Model-based Development . Technical Report MSR-TR-2008-86, Microsoft Research (June 2008)
10. Benveniste, A., Caillaud, B., Carloni, L.P., Caspi, P., Sangiovanni-Vincentelli, A.L.: Composing heterogeneous reactive systems. ACM Trans. Embedded Comput. Syst. **7**(4) (2008)
11. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Workshop on Generative Techniques in the Context of Model-Driven Architecture (OOPSLA). (2003) 1–17

12. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: An Integrated Electronic System Design Environment. IEEE Computer **36**(4) (2003) 45–52
13. Weisemöller, I., Schürr, A.: Formal Definition of MOF 2.0 Metamodel Components and Composition. In: Proceedings of the Model Driven Engineering Languages and Systems. (2008) 386–400
14. Neema, S., Sztipanovits, J., Karsai, G., Butts, K.: Constraint-Based Design-Space Exploration and Model Synthesis. In: Proceedings of the International Conference on Embedded Software (EMOFT). (2003) 290–305
15. Czarnecki, K., Wasowski, A.: Feature Diagrams and Logics: There and Back Again. In: Proceedings of the International Conference on Software Product Lines (SPLC). (2007) 23–34
16. Dantsin, E., Voronkov, A.: Expressive Power and Data Complexity of Query Languages for Trees and Lists. In: Proceedings of the Symposium on Principles of Database Systems (PODS). (2000) 157–165
17. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). (2008) 337–340
18. Jackson, D.: Alloy: A New Technology for Software Modelling. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (2002) 20
19. Object Management Group: Mda guide version 1.0.1. Technical report (2003)
20. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: a time-triggered language for embedded programming. Proceedings of the IEEE **91**(1) (2003) 84–99
21. Kopetz, H., Bauer, G.: The time-triggered architecture. Proceedings of the IEEE **91**(1) (2003) 112–126
22. Pree, W., Templ, J.: Modeling with the Timing Definition Language (TDL). In: Workshop on Model-Driven Development of Reliable Automotive Services (ASWSD). (2006) 133–144
23. Gurevich, Y., Neeman, I.: DKAL: Distributed-Knowledge Authorization Language. In: 21st IEEE Computer Security Foundations Symposium (CSF). (2008) 149–162
24. Becker, M.Y., Fournet, C., Gordon, A.D.: Design and Semantics of a Decentralized Authorization Language. In: 20th IEEE Computer Security Foundations Symposium. (2007) 3–15