# A Case Study in Using Model-Based Concepts for Engineering Component-Based Software Systems

Daniel Welch and Murali Sitaraman

**Technical Report RSRG-16-04**
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

June 2016

Copyright © 2016 by the authors. All rights reserved.

**manuscript No.**
(will be inserted by the editor)

# A Case Study in Using Model-Based Concepts for Engineering Component-Based Software Systems

**Daniel Welch** · **Murali Sitaraman**

the date of receipt and acceptance should be inserted later

**Abstract** The aim of this paper is to illustrate the central role of concepts based on mathematical models in decoupling components and modularizing software systems. As systems scale in terms of the number of components and the inherent complexity of a given domain, the ability to formally state concepts precisely and concisely in terms of mathematical models and develop alternative components to realize these same concepts becomes increasingly important. This paper describes in detail a component-based case study in which all components are built for reuse and have interface contracts that specify concepts formalized via mathematical models. The paper outlines a component-based software engineering case study that brings together a number of novel, model-based concepts, each with formal interface contracts and alternative component implementations. The concepts, components, and the system are set up to be amenable to automated, modular verification.

**Keywords** components · contracts · mathematical modeling · modular design · reuse · verification

## 1 Introduction

While the need for well-engineered component-based software in improving software productivity and quality is widely acknowledged, the benefits are significantly enhanced if the engineering is performed in a model-driven fashion. While such models are often application (and domain)-specific, this paper extends these principles to reusable concepts that can be described precisely using mathematical models which enhance generality and applicability, facilitate alternative component implementations with varying performance behaviors, and most of all, serve as the basis for formal analysis that can lead to automatically verified components and component-based software.

The paper presents a case study involving a variety of novel concepts. The concepts presented are all generic, and each is explained with a formal interface contract that uses a precise mathematical model to specify functional behaviors. Unlike contracts that are often tightly coupled to the particulars of the implementations/code they attempt to describe, the contracts here are abstractly described via purely mathematical models, and are therefore devoid of implementation bias.

The contracts describing the concepts hide not only what structures are used in underlying computations, but also when such computations take place. Consequently, each concept described in the paper is shown to have a host of component implementation possibilities. Though not a focus of this paper, the conceptual models presented can be extended to specify non-functional performance behaviors of alternative component implementations.

A key feature of the components described in the paper is that each one is typically implemented by assemblies of other concepts and components. This layered construction is crucial for enabling formal and automated modular analysis of component-based software, one component at a time [22]. To facilitate automated analysis, the component implementations are annotated with internal assertions, such as invariants.

The overall contribution of this paper is that it shows the benefits of model-based engineering of component-based software with a non-trivial case study involving a host of concepts, described using mathematical models and component assemblies—all of which are amenable to automated analy-

This research has been funded in part by the US NSF grants CCF-0811748, CCF-1161916, and DUE-1022941

Daniel Welch
dtwelch@clemson.edu

Murali Sitaraman
murali@clemson.edu

sis and verification. The rest of the paper is organized into the following sections. Section 2 presents the overall architecture, showing the relationships among the artifacts involved in the case study. Section 3 presents summaries of concepts, their mathematical models, and a variety of interchangeable component assemblies for each. Sections 4 through 6 illustrate the development of one novel concept in detail, including a mathematical theory built to simplify and abstract the concept's description. Section 7 contains work most closely related to the case study and a summary.

## 2 Overview of a component-based case study

To illustrate the notion of formulating generalized concept interfaces from specific examples, and to better explain the role mathematical modeling plays in engineering such concepts and their component-based implementations, this section considers a well-known optimization problem in the domain of graph algorithms:

- An algorithm to find a minimum spanning tree in a (connected), edge-weighted graph.[1]

Unlike the algorithm which is coupled to a particular data structure to solve one specific problem, the generalization of the problem yields a reusable concept that is:

- Generalized so that optimization is based on some general function of edge information.
- Modeled so that it allows for incremental delivery of edges of interest, and extraction of a subset of minimum spanning edges (including, of course, the entire set).
- Designed so that it is suitable even when the input graph is not connected (in which case the solution is said to be a *minimum spanning forest*).

The generalization of weighting based on edge information allows possibilities such as, for example, traffic times in a graph of streets to be affected by a variety of factors like geography or weather conditions—rather than just distances. The key notion of *incremental* construction is motivated by efficiency, generality, and reuse concerns [26]: for example, there are many applications in which clients of such an interface might not wish (or need) to process/obtain all edges of the resulting spanning forest, such as when a fixed total threshold bound can be met. The importance and utility of such a concept being able to scale to the problem of unconnected networks, thus producing forests as solutions is obvious—thus the resulting interface contract design for the concept should accommodate these considerations. We term this reusable concept a `Spanning_Forest_Finder` and discuss it in the next subsection.

[1] Informally, a minimum spanning tree is defined to be a subset of a graph's weighted-edges such that all vertices are connected with minimum total weight.

A layered, component-based architecture using a variety concepts to realize a solution to spanning forest finding problem is given in Figure 1. For this solution, the figure shows the use of two other concepts: one that captures the idea of connectivity, and one that captures the idea of prioritization. Solutions to the prioritization problem introduce yet another layer that utilizes a new spiral like concept. While the remainder of this section is largely reserved for discussion of the mathematical modeling of the spanning forest finding concept, we also summarize the models of additional concepts instantiated and used in the context of Kruskal-based implementation. Finally, discussion of the spiral component (appearing in layer 3) is given special treatment later on in section 4, and is therefore is best examined after prioritization is discussed.

Together, the concepts illustrate the central role of a range of mathematical models, including common ones, such as graphs and multisets, to occasionally new, more exotic ones, such as spirals. The models make it possible to flexibly define formal interface contracts that facilitate automated analysis and verification.

### 2.1 Layer 1: A concept for building minimum spanning forests

An interface contract for the spanning forest concept is shown in listing 1 in RESOLVE [22], a language especially designed to facilitate reusable conceptual model specifications in conjunction with component implementations that can be suitably annotated for verification. These ideas can be expressed, though to a limited extent, in other modeling, programming language-dependent (e.g., JML [15,14]) or independent formalisms (e.g., Event-B[1,16] or Z [24]).

```
Concept Spanning_Forest_Finder(
  type Edge_Info,
    evaluates Vertex_Max : Integer;
    Def Weight(El : Edge_Info) : ℕ);

   Type family Edge ⊆ Cart_Prod
      u, v : ℕ;
      Label : Edge_Info;
   end;
   exemplar E;
   constraints
      0 ≤ E.u, E.v ≤ Vertex_Max and
            E.u ≠ E.v;

   Type family Graph_Holder ⊆ Cart_Prod
      Graph : 𝒫(Edge);
      In_Insertion_Phase : 𝔹;
```

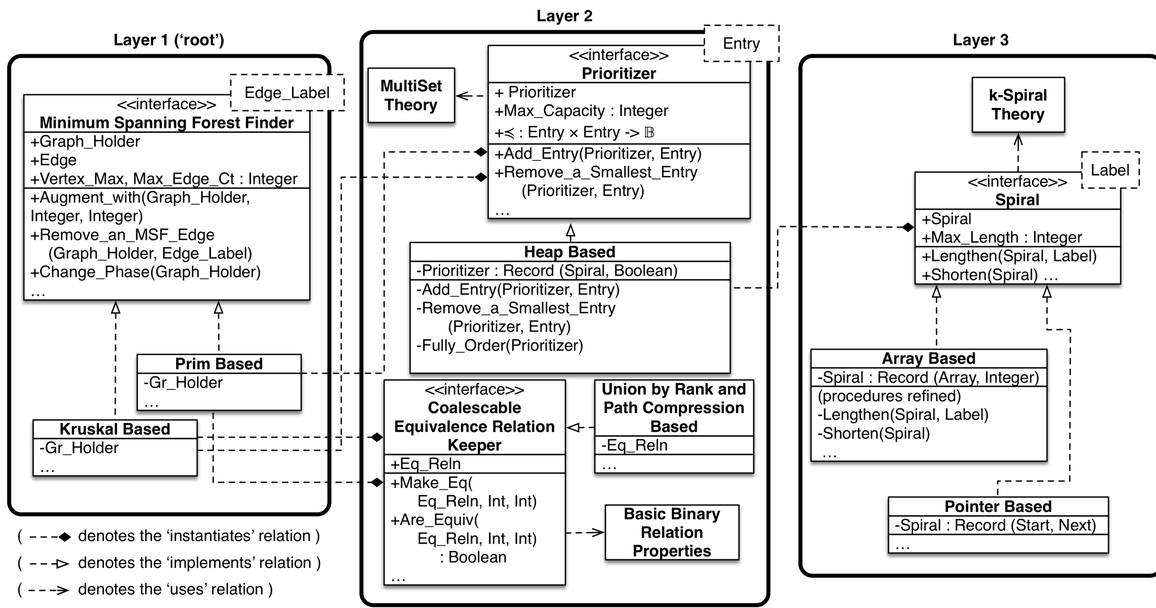


**Fig. 1** A UML class diagram of the proposed layered, component-based architecture. Each layer is encircled by boxes drawn with solid lines.

```
    end;
    exemplar GH;
    constraints
        ¬ GH.In_Insertion_Phase implies
                    Is_MSF(GH.Graph);
    ...
    Operation Augment_with, Change_Phase,
        Remove_an_MSF_Edge(..);
    ...
end Spanning_Forest_Finder;
```

**Listing 1** A partial concept interface for the spanning forest finder

### 2.1.1 Mathematical modeling

Since the focus of this paper is on conceptual modeling, we emphasize mostly the modeling and not the details of specific operations or their implementations. In teasing out a suitable mathematical model based on a graph structure, first, we model the notion of an `Edge` as the cartesian product (**`Cart_Prod`**) of natural numbers `u`, `v`, and a generic type `Edge_Info`. The numbers `u` and `v` simply represent 'tags' for the vertices bookending the edge, while the weight used in optimization is based on `Edge_Info`, and is some application-dependent function of `Weight` supplied as a parameter by the concept user at the time of instantiation.

The **`exemplar`** clause gives specifiers a handle with which to refer to an arbitrary `Edge`, and can subsequently use it to impose some **`constraints`** on the abstract type: namely that the natural numbers forming the 'ends' of the edge fall between 0 and `Vertex_Max`, and that they are indeed not equal—which, for the sake of simplicity, disallows self loops in the resulting graphs.

With the notion of an `Edge` abstractly defined, we now turn to defining a suitable model for the graph that takes into consideration the aforementioned two-phase design considerations of the component. Thus, we include in the `Graph_Holder` model not only the set of edges that make up the graph (formulated as a member of $\mathscr{P}$(`Edge`)), but also a boolean flag `In_Insertion_Phase` which indicates the phase the machine is currently operating under: insertion or extraction. The key **constraints** (i.e., *conceptual invariants*) asserts via the `Is_MSF` predicate that when the finder is not in the insertion phase, the graph it holds only contains edges in a minimum spanning forest of the input graph. Of course, this is just an abstraction: that such a component implementation (detailed in this paper) can indeed compute the MSF incrementally as and when edges are extracted—and that implementation indeed needs an abstraction relation for verification—is the topic of [23].

### 2.1.2 Concept interface design to provide functional and performance flexibility

The interface operations for the concept—rather than being structured around a single large effect algorithm (such as `Find_MSF`, which under the hood would simply invoke Prim or Kruskal's MST finding algorithm)—are instead designed to be "small effect" (incremental) in nature and as such, are centered around the notion of a two phase machine. That is, by repeatedly calling the `Augment_with` operation, clients are free to add the edges they are concerned

with to the graph, and, when finished, can simply switch the machine into extraction mode by invoking the Change_Phase operation then retrieve edges of a minimum spanning forest, one at a time, via the Remove_an_MSF_Edge operation. This approach offers the greatest amount of flexibility to both clients (because they no longer have to pay to compute an entire MSF in bulk if not needed for their application – though they can still do so if they wish) as well as implementers (because they can choose when and where edge prioritization takes place). None of these implementation or usage possibilities affects the contracts or mathematical models.

## 2.2 Layer 2: Concepts for prioritizing and checking connectivity

A greedy solution to the spanning forest finding problem (due to Kruskal [11]) hinges on the ability to efficiently perform two tasks: (1) order edges with a weighting function based on edge information and (2) test whether adding an edge to the under-construction forest produces a cycle. In devising a suitable solution for the optimization problem, we seek other independent, reusable concepts that are useful—not just for solving the spanning forest problem using Kruskal's approach—but those applicable to a whole class of connectivity and prioritization problems. Such are the concepts appearing in layer two of the architecture in Figure 1. In this section we give a high level overview of the conceptual models for describing these problems.

```
Definition Is_Lighter_Weight_Edge
        (E1, E2 : Edge) : 𝔹 =
             (Weight(E1) ⩽ Weight(E2));
```

**Listing 2** A predicate that defines edge ordering in terms of the Weight function.

To achieve the objective of ordering edges, a Kruskal component implementation (in layer 2 of Figure 1) would simply supply the Is_Lighter_Weight_Edge definition to the prioritizer concept interface appearing in listing 3[2]. Such a definition is naturally dependent on the Weight function given to the spanning concept interface contract.

### *2.2.1 Prioritizing concept*

Used to order the edges of the graph being augmented, the interface contract for the prioritizer concept—a sketch of which appears in listing 3—is parameterized by a generic Entry (which in our case is specialized by the Edge type model) in addition to a total binary preordering relation[3] $\preccurlyeq$ which dictates how the edges should be ordered.

```
Concept Prioritizer_Template(type Entry;
   Max_Capacity : Integer;
     Def (x : Entry) ≼ (y : Entry) : 𝔹);
  uses Multiset_Theory;

  Type family Prioritizer ⊆ Cart_Prod
    Entry_Tally : FMSet(Entry);
    Is_Accepting : 𝔹;
  end;
  exemplar P;
  constraints
        ||P.Entry_Tally|| ⩽ Max_Capacity;
  initialization
       ensures P.Is_Accepting and
                    P.Entry_Tally = Φ;
   ...
  Operation Add_Entry, Change_Phase,
     Remove_a_Smallest(...);
  ...
end Prioritizer_Template;
```

**Listing 3** A partial interface contract for the prioritizer concept

Like the spanning forest interface, the organization of the prioritizer also conforms to the aforementioned machine oriented design principle which lends similar advantages both in terms of understandability and reuse, as well as implementation and performance flexibility, as discussed in section 2.1.2. [4]

### *2.2.2 Mathematical modeling*

Operating at the core of the prioritizer's mathematical model is a multiset (also sometimes referred to as a 'bag') with relevant notions defined in the imported Multiset_Theory. The **constraints** in this case dictates that the cardinality of the Entry_Tally multiset do not exceed the bounds given, while the **initialization** clause immediately below simply states what can be expected by clients when an instance of the Prioritizer is declared: namely that it is accepting new entries, and that the 'tallying' field is equal to the empty multiset, $\Phi$—formally defined in Multiset_Theory.

[2] In terms of types, the $\preccurlyeq$ relation parameterizing the prioritizer interface contract carries the highly general type schema: Entry $\times$ Entry $\to \mathbb{B}$. And since Edge in this case *is* the concrete specialization of Entry, the Is_Lighter_Weight_Edge (having type Edge $\times$ Edge $\to \mathbb{B}$) is perfectly acceptable from a type perspective

[3] Put another way, the prioritizer contract requires that the relation supplied is total *and* transitive. It is also important to note that this parameter is a mathematical definition, not an actual operation for ordering, which would be required to implement the concept and must be supplied by prioritizer client.

[4] Though the present application of the prioritizer concept for the spanning forest finder only requires a change from insertion to extraction phase, the concept is more general and allows changing phases to allow interleavings of entry insertions and extractions

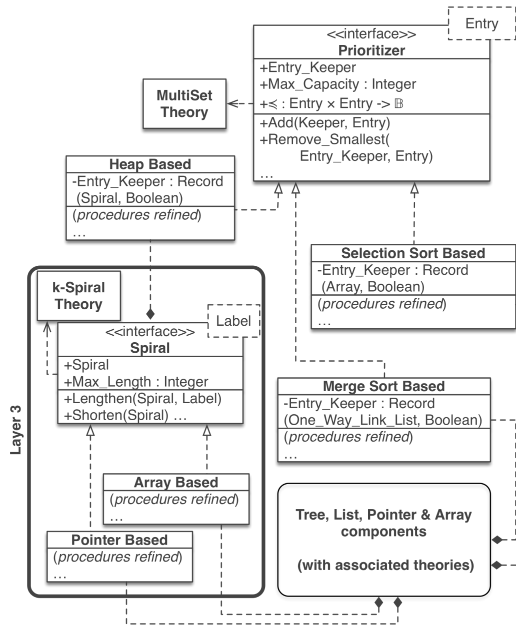


**Fig. 2** An enriched perspective of layer 3 illustrating some alternative implementations of the prioritizer concept.

Figure 2 enriches the original architectural diagram with an additional view of the prioritizer concept, emphasizing its many interchangeable (though functionally equivalent) implementations. Among these is the heap-based component implementation that is the topic of Section 4.1.

#### *2.2.3 A coalesceable equivalence relation concept*

While the ability to check connectivity is one particular problem in finding spanning forests, the more general concept is one that captures arbitrary equivalences. Loosely speaking, 'objects' begin unconnected in their own equivalence classes, and adding a new equivalence between two objects collapses their corresponding classes which might connect them – leading to a cycle. The general concept to form and maintain equivalence relations is the topic of listing 4.

```
Concept Coalesceable_Eqv_Relation(
   Max_Size : Integer);

  Type family Eq_Reln ⊆ ([1..Max_Size] ×
          [1..Max_Size]) → 𝔹;
    exemplar E;
    constraints Is_Reflexive(E) and
          Is_Symmetric(E) and
          Is_Transitive(E);
    initialization
        ensures ∀ x, y : [1..Max_Size],
              E(x, y) implies (x = y);
  Operation Are_Equivalent,
     Make_Equivalent(..);
     ...
end Coalesceable_Eqv_Relation;
```

**Listing 4** A partial interface contract for an equivalence relation concept

To ensure that instances of equivalence relation type conform with the mathematical modeling, the concept makes use of several predicates from the `Basic_Binary_Relation_Properties` theory. Specifically, the **constraints** ensure that the model is indeed reflexive ($\forall x.x\rho x$), symmetric ($\forall x,y.x\rho y \Rightarrow y\rho x$), and transitive ($\forall x,y,z.x\rho y \wedge y\rho z \Rightarrow x\rho z$), while the **initialization** clause ensures that all indices are initially disjoint (in other words, that each initially inhabits its own equivalence class).

The set of core operations allow clients to either (1) form an equivalence relation between two indices `i` and `j` via a call to `Make_Equivalent`, (2) test whether `i` and `j` inhabit the same equivalence class by calling `Are_Equivalent`, or (3) reset all existing classes to their initial state (thus deleting any and all relationships among the indices) by invoking the `Clear` operation.

Once again, irrespective of the formal contract for the concept laid out above, clients are free to implement it in any number of efficient ways including (but not limited to) approaches utilizing the well-known 'union by rank' and 'path compression' optimizations [6].

## 3 Component-based architecture layer three: motivating the need for additional abstraction

Having detailed the concepts within the first two layers of the solution architecture, in this section we motivate and detail the spiral concept that underlies a heap-based component implementation of the prioritizer, which appears in Figure 2.

While the example concept is a specific one, the purpose of this section is in elucidating the ideas in developing any new concept using a new mathematical model, especially developed to explain that concept. For this reason, this section is quite detailed and begins with an informal explanation to provide an intuitive understanding of the ideas, and concludes with a formal presentation of both an interface contract for the concept, as well as a concrete implementation suitable for automated verification. While the other concepts in this paper could be further elaborated as well, we refrain from doing so in the interest of keeping the discussion manageable.

The heap-based component implementation of a prioritizer in particular enjoys certain performance benefits over the alternatives due to the complete binary tree-like structure of heaps, which, when paired with a *heap property*[5] makes it

[5] A property on heaps which ensures the root of any particular subtree contains the smallest (or largest) element among its children

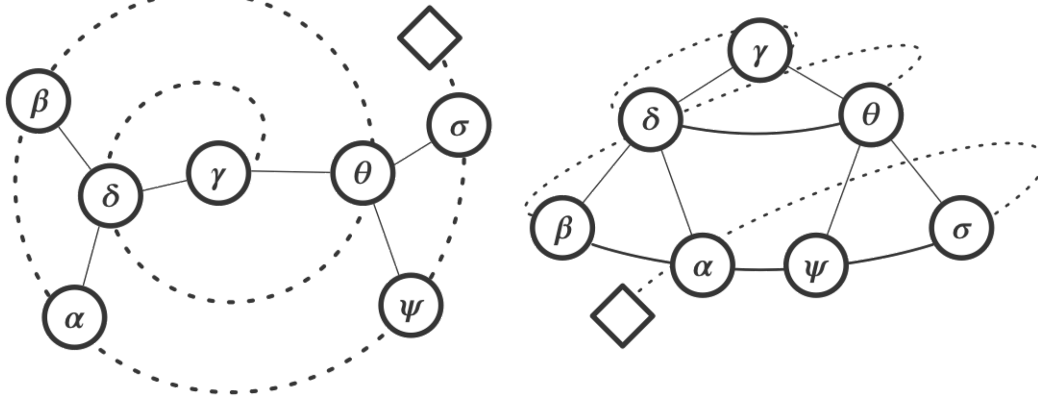


**Fig. 3** Two perspectives of an arity $k = 2$ spiral. Left is a top down (stovetop) perspective, while the right shows the same spiral from a different (tower-like) perspective.

possible to access, delete, and insert elements efficiently [6]. Moreover, the choice of heap is particularly amenable to the machine-oriented, incremental design of the prioritizer, as each time a user calls `Add_Entry`, it can simply be added to the top of the heap, then 'trickled' down into the correct position based on the generic $\preccurlyeq$ relation passed–thus offsetting the need for a bulk sorting computation to take place in `Change_Modes`.

From a component-based design perspective, the task of implementing and authoring concise, readable (much less automatically verifiable) formal contracts for a heap based implementation of the prioritizer entails contending with one of two relatively unattractive design options:

1. The first is to simply instantiate and use a binary tree—thereby committing the implementation to explicitly keep invariant (in terms of contracts *and* code) the required balance and heap ordering properties, which is clearly not preferable in terms of automation or ease of reasoning.
2. The second approach is to simply 'fall back' to the defacto, 'textbook' standard and use an array-based representation of the heap. This however degrades two important attributes:
   – *abstraction*–one must map the notion of the tree onto the raw indices of the array, which in turn complicates the contracts and hence the overall reasoning process.
   – *understandability*–one of the primary advantages of mathematical modeling is having an explicit, well defined conceptual model to work with and write contracts in terms of—this approach precludes these benefits.

In other words, using the array directly is insufficient from a conceptual modeling perspective, as it lacks a suitable conceptual veneer. In the remaining sections, we propose an alternative design that gives rise to a new reusable concept that provides an additional layer of abstraction.

## 4 Model-based engineering of a new concept, Part 1: An informal introduction the *k*-spiral concept

In place of the rooted tree conceptualization and its typical raw-array based representation, the spiral concept employs a system of linked locations, connected up in a spiral-like configuration. Figure 3 illustrates an example spiral of arity $k = 2$ whose locations are labeled by greek letters and linked together via arcs of the spiral starting from the fixed central location, $\gamma$, and terminating at the diamond shaped location. Assuming the existence of a current location cursor—which for illustration purposes we fix to start at $\gamma$—conceptually one can think of cursor traversal throughout the spiral in three different ways including (1) a linear scan-through following the arcs of the spiral: $\gamma, \delta, \theta, \beta, \ldots, \sigma$, (2) vertically along the directed links connecting one arc level to another: $\gamma, \delta, \beta$, and (3) combining the two for a cyclic style traversal such as $\gamma, \delta, \theta$.

```
Concept Spiral_Template
    (type Label;
     k, Max_Length : Integer);
 Type family Spiral;
     /* is modeled by ... */

 Oper Lengthen(S : Spiral; e : Label);

 Oper Shorten(S : Spiral; e : Label);

 Oper Spiral_Out(S : Spiral);
 Oper Spiral_In(S : Spiral);

 Oper Hop_Out(S : Spiral);
```

```
 Oper Hop_In(S : Spiral; e : Integer);

 Oper Move_to_End(S : Spiral);
 Oper Move_to_Center(S : Spiral);

 Oper Swap_Label(S : Spiral; e : Label);
 Oper Length_of(
     S : Spiral_Pos): Integer;

 Oper Clear(S : Spiral);
   ...
end Spiral_Template;
```

**Listing 5** Interface contract for a spiral concept.

An abbreviated, interface contract for the spiral concept including a set of primary operations appears in listing 5. The interface given is parameterized by a generic type, `Label`, allowing the structure to store objects of an arbitrary type. The `k` parameter specifies the desired arity of spirals produced by this concept (meaning how many 'sub-locations' are directly reachable from each location along the arcs) while the `Max_Length` parameter serves as the upper bound on the length spirals can have. The interface exports a single type, `Spiral`—as introduced by the **`Type family`** clause.

The core operations allow clients to add, remove, relabel, and otherwise traverse over the locations that make up the conceptual `Spiral` model. Figure 4 provides an illustration of the various modifications these operations permit on an example spiral `S`, such as building a spiral over successive calls to `Lengthen`, or moving the current location cursor either (1) linearly along the arcs (via 'spiraling' in-/out) or (2) vertically (via 'hopping') from one radial arc level to another.

### 4.1 Instantiating and using the spiral in a heap-based prioritizer implementation

Like any other component in RESOLVE, before this particular concept can be used, it must first be instantiated via a facility declaration, and provided with arguments matching the formal interface parameters shown in listing 5.

```
Implementation Heap_Impl(
   Operation Is_GTR(x, y : Entry) : 𝔹;
      ensures Is_GTR = (¬ x ≼ y);
   for Prioritizer_Template;

  Facility Sp_Fac is Spiral_Template
        (Entry, 2, Max_Capacity);
    implemented by Sp_Array_Impl;

  Type family Prioritizer = Record
    Heap : Sp_Fac :: Spiral;
    Accpt_Flag, Fully_Ord_Flag : Boolean;
  end;

  /* An internal (private) operation */
  Oper Fully_Order(P : Prioritizer);
   ensures ...
    Proc
      Var i : Integer;
      If P.Fully_Ord_Flag = false then
      Move_to_End(P.Heap);
      Iterate
         maintaining ...
         When At_Center(P.Heap) do exit;
         Hop_In(P.Heap, i);
         Fix_Pos(P.Heap);
      Repeat;
      P.Fully_Ord_Flag := true;
   end;
  end Fully_Order;

  Proc Add_Entry(
      P : Prioritizer; e : Entry);
   ...
  end Add_Entry

  Proc Change_Phase(P : Prioritizer);
    If P.Accpt_Flag then
      Fully_Order(P);
    end;
    P.Accpt_Flag := not P.Accpt_Flag;
  end Change_Phase;

  Proc Remove_a_Smallest(
      P : Prioritizer; e : Entry);
    Shorten(P.Heap, e);
    Move_to_Center(P.Heap);
    Swap_Label(P.Heap, e);
    Fix_Pos(P.Heap);
  end Remove_a_Smallest;
    ...
end Heap_Impl;
```

**Listing 6** Instantiating the spiral concept in the context of (an abridged) heap-based prioritizer component implementation.

In the context of a heap-based implementation of the prioritizer—shown in listing 6, instantiation entails supplying a label type (which in this case happens to be another generic `Entry`), an arity $k = 2$, and the maximum possible length spirals drawn from the facility can take on—in other words, the number of entries the prioritizer concept itself is capable of holding—denoted by `Max_Capacity`.

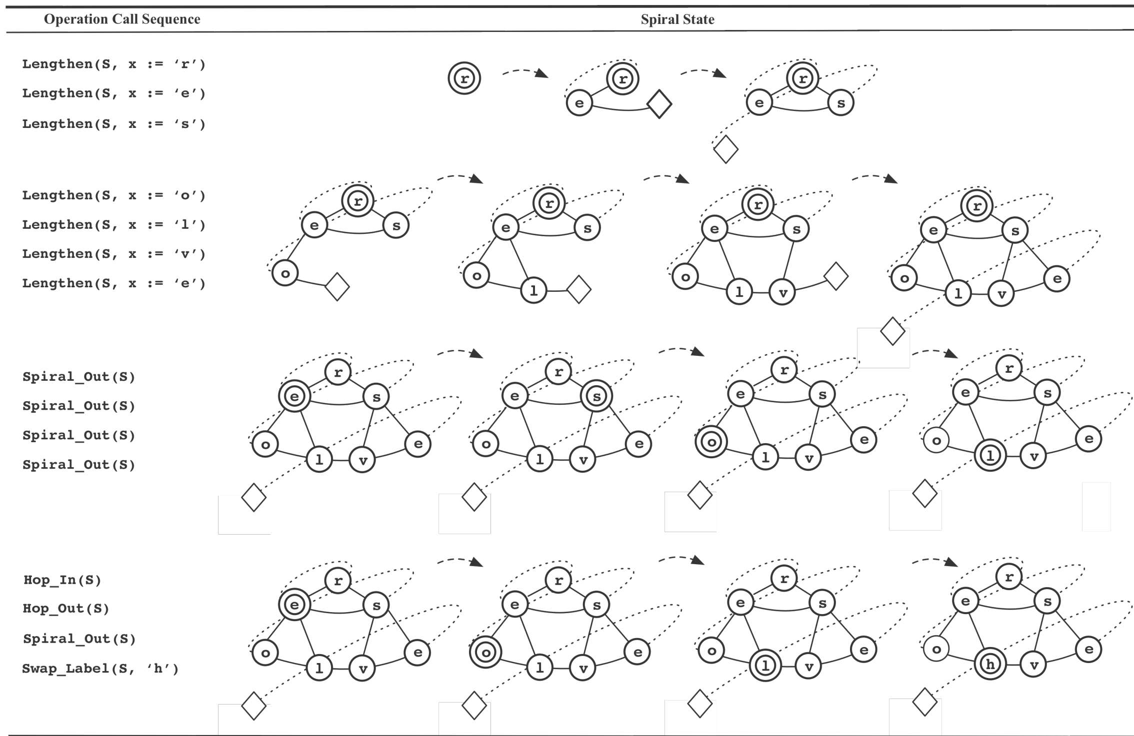


**Fig. 4** An 2-ary spiral built up via repeated calls to `Lengthen`, where `S` and `x` are of type `Spiral` and `Character`, respectively. The double-circles indicate the position of the current location cursor.

Using the `Sp_Fac` facility declaration, which pairs the interface of our conceptual spiral (`Spiral_Template`) with an array-based implementation (`Sp_Array_Impl`), we programmatically represent the prioritizer model discussed in section 2.2.2 as a record containing two fields: a spiral instance `Heap` (drawn from the `Sp_Fac` facility) used in lieu of an array, and two booleans: `Accpt_Flag` and `Fully_Ord_Flag` that are responsible for tracking which phase the machine is currently operating under (extraction or insertion) and whether or not the `Heap` conforms with the heap property, respectively. The `Heap` field within the record can be referenced (and modified) by any procedure in the implementation of the prioritizer so long as it has access to some variable or parameter `P` of type `Prioritizer`.

Without needing to worry about the balance of the heap (as `Lengthen` automatically preserves this), all logic for maintaining the $\preccurlyeq$-dependent heap property can be written exclusively in terms of spiral operations. For example, `Fully_Order` is an internal operation used to convert a (potentially) non $\preccurlyeq$-conformal `P.Heap`[6] into one that conforms with this property. The code presented essentially performs the same steps one would normally expect to see in the standard textbook implementation of such a routine: That is for each (arc) level we iteratively hop up to the location on the parenting (arc) level until we reach the root (center), each time invoking the internal recursive operation `Fix_Pos`—which is a spiral-based equivalent of the textbook 'heapify' routine [6].

[6] The '.' is reserved exclusively for program record (or mathematical cartesian product) field accesses, while the '::' is used for facility (or module level) scope accesses

## 5 Model-based engineering of a new concept, Part 2: Mathematical developments for a new *k*-spiral theory

In order to specify formally the informal contract in the interface listing 5 for the spiral concept, we first must define a suitable mathematical model from a domain expressive enough to capture the structure of the spiral itself and the intended behavior of its operations, while remaining general enough to permit reasoning about spirals of an arbitrary arity. Even with a mathematical framework meeting such criteria, overriding design concerns include readability, simplicity, and the verifiability of the resultant contracts. This mathematical development is the topic of 5.1.

### 5.1 *k*-spiral theory preliminaries

In order to write contracts for `Spiral_Template`, there should be a well defined mathematical domain from which to draw notations, facts, and results. While, in general, the

need for new theories should be few and far between (as existing theories should be reused as much as possible), the spiral example represents a compelling exercise in developing a new, generalized theory. `Basic_Spiral_Theory` is a mathematical unit for encapsulating a variety of developments ranging from definitions and theorems, to corollaries useful in authoring succinct, readable contracts for the spiral (or any other concept interfaces for that matter in need of a complete *k*-ary graph like structure). In the interest of maintaining a separation of concerns, proofs of these results are relegated to a separate module that may be verified offline using popular proof assistants such as Isabelle[18] or PVS[20].

All functions needed to categorically describe the inductive structure of *k*-spirals are encapsulated together by the **categorical** definition in listing 7.[7]

```
Categorical Definition for
   Sp_Loc : (k : ℕ^≥2) → SSet,
   Cen : (k : ℕ^≥2) → Sp_Loc(k),
   SS, RS :
     (k : ℕ^≥2) → Sp_Loc(k) → Sp_Loc(k)
  is Is_k_Spiral_Like
       (Sp_Loc, Cen, SS, RS);
```

**Listing 7** *k*-spiral functions encapsulated within the context of a categorical definition.

More than just a mechanism for bundling together various functions, the predicate following the **is** keyword allows specifiers to relate these seemingly disparate pieces via user defined predicates like `Is_k_Spiral_Like`—which in this case is used to express the core axiomatic properties of the mathematical structure being built.

While a full, formal explanation of this predicate is outside the scope of this paper, the definitions introduced in listing 7 (and later passed into the predicate) form the basis of the spiral structure, and can be read as follows:

- Spiral location (`Sp_Loc`): a function that, given some suitable value of *k*, produces a location in the spiral. Intuitively the notion of a 'spiral location' can be thought of as the circles in the spiral diagrams presented thus far (such as that in Figure 3).
- Spiral center (`Cen`): the location representing the first, central location of a given spiral.
- Spiral, radial successor (`SS`, `RS`): successor functions that enable transition from location to location either outwards (from the center) along the arcs (*SS*) or outwards across radial arc layers (*RS*).

**Definition 1 (Spiral Center Distance)**

`SCD(p : Sp_Loc(k : ℕ^≥2)) : ℕ`

The spiral center distance (*SCD*) of some location *p* is the number of successor applications (*SS*) needed to reach *p* from the center, proceeding outwards along the arcs.

```
Theorem Sp1: ∀ k : ℕ^≥2, ∀ p : Sp_Loc(k),
     SCD(p) < SCD(RS(k)(p));
```

**Listing 8** An example theorem from *k*-spiral theory

For illustration, an example of a theorem (or result) from the theory is shown in listing 8. There are many others like this one for use by automated verification systems.

**Definition 2 (Inward Location)**

```
Inward_loc(b : Sp_Loc(k : ℕ^≥2)) :
                      𝒫(Sp_Loc(k))
```

The inward location of a given location *b* is defined to be the set of locations 'preceding' *b* in the spiral – where a location *u* is said to precede *v* if $SCD(u) < SCD(v)$. Figure 5 illustrates an application of this function.

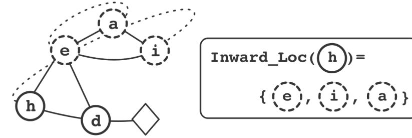


**Fig. 5** Applying `Inward_Loc` to the location labeled '*h*'.

**Definition 3 (Radial Predecessor)**

`RP : (k : ℕ^≥2) × Sp_Loc(k) → Sp_Loc(k)`

Defined to be the complement of radial successor (*RS*) function in listing 7, this function produces a location's 'parent' situated one arc level inward—except for the center, in which case $RP(k)(Cen(k)) = Cen(k)$. In terms of the example spiral in Figure 5, the radial predecessor of the location labeled by 'h' is the location labeled 'e'.

**Definition 4 (In Sector of)**

```
(q : Sp_Loc(k : ℕ^≥2)) In_Sect_of
       (p : Sp_Loc(k)) : 𝔹
```

A location *q* is said to be in *p*'s sector if there exists some $n \in \mathbb{N}$ such that the radial predecessor function (*RP*) applied *n* times from *q* produces *p*.

## 6 Model-based engineering of a new concept, Part 3: A formal interface contract and an implementation of spirals

Having the spiral theory and its included definitions at our disposal, a formal interface contract (originally outlined in listing 5) for a spiral concept is given in Figure 6.

[7] In ascii, we represent the set $\mathbb{N}^{\geqslant 2}$ with the following additional binding: **Def** `N2` : $\{n : \mathbb{N} \mid n \geqslant 2\}$

```
Concept Spiral_Template(
  type Label; evaluates k,
        Max_Length : Integer);
 uses ..., Basic_Spiral_Theory;
 requires k ⩾ 2 and Max_Length > 0
                  which_entails k : ℕ^{⩾2};

 Type family Spiral ⊆ Cart_Prod
     Lab : Sp_Loc(k) → Label;
     Curr_Loc, Trmnl_Loc : Sp_Loc(k);
 end;
 exemplar S;
 constraints
  SCD(S.Trmnl_Loc) ⩽ Max_Length and
  S.Curr_Loc ∈ Inward_Loc(S.Trmnl_Loc) and
  S.Lab(S.Trmnl_Loc) = Label.base_point and
  S.Lab[Sp_Loc(k) ∼ Inward_Loc(S.Trmnl_Loc)] =
     {Label.base_point}
 initialization
     ensures S.Trmnl_Loc = Cen(k);

 Operation Lengthen(
        updates S : Spiral;
              alters e : Label);
  requires SCD(S.Trmnl_Loc) < Max_Length;
  ensures S.Trmnl_Loc =
        SS(k)(#S.Trmnl_Loc) and
    S.Lab = #S.Lab except
        S.Lab(#S.Trmnl_Loc) = #e;

 Operation Spiral_Out(
        updates S : Spiral);
  requires SS(k)(S.Curr_Loc) ∈
              Inward_Loc(S.Trmnl_Loc);
  ensures S.Curr_Loc = SS(k)(#S.Curr_Loc);

 Operation Hop_Out(updates S : Spiral);
   requires RS(k)(S.Curr_Loc) ∈
              Inward_Loc(S.Trmnl_Loc);
   ensures S.Curr_Loc = RS(k)(#S.Curr_Loc);

 Operation Move_to_End(updates S : Spiral);
   ensures S.Curr_Loc = #S.Trmnl_Loc;
...
end Spiral_Template;
```

```
Implementation Sp_Array_Impl
                 for Spiral_Template;

 Type Spiral = Record
     Labl : Array 0..Max_Length - 1 of Label;
     Length, Curr_Place : Integer;
  end;
  convention
   0 < S.Curr_Place and
          S.Curr_Place ⩽ S.Length and
                S.Length ⩽ Max_Length
        which_entails
              S.Curr_Place, S.Length : ℕ;
  correspondence
   conc.S.Trmnl_Loc = SS(k)^{S.Length}(Cen(k)) and
   conc.S.Curr_Loc = SS(k)^{S.Curr_Place}(Cen(k)) and
      λ q : Sp_Loc(k).
    ({ S.Labl(SCD(q)) if SCD(q) + 1 ⩽ S.Length
     { Label.base_point      otherwise;        );

 Procedure Lengthen(
        updates S : Spiral;
              alters e : Label);
   S.Labl[S.Length] :=: e;
   S.Length := S.Length + 1;
 end Lengthen;

 Procedure Spiral_Out(
        updates S : Spiral);
   S.Curr_Place := S.Curr_Place + 1;
 end Spiral_Out;

 Procedure Hop_Out(
        updates S : Spiral);
   S.Curr_Place := k * S.Curr_Place + 1;
 end Hop_Out;

 Procedure Move_to_End(
        updates S : Spiral);
   S.Curr_Place := S.Length;
 end Move_to_End;
...
end Sp_Array_Impl;
```

**Fig. 6** A complete formalized spiral interface contract (left) and an array-based implementation (right).

### 6.1 Formalizing the concept

The spiral type is modeled mathematically as the cartesian product of three fields: A labeling function `Lab`, and two distinguished locations: `Curr_Loc` and `Trmnl_Loc` that point to a current and last location in the spiral, respectively. The four **`constraints`** on this particular model (split in order of the conjuncts, from first to last) are summarized as follows:

- The spiral center distance of the last location is within the bounds of `Max_Length`.
- The current location is included in the inward location set of the terminal location.
- The `Label` of the terminal location is of type **`base_point`**: a special type in the language indicating that *any* value of type `Label` is permitted, including, of course its initial value.
- All spiral locations outside of the current spiral (of which there are infinitely many) map to `Label`.**`base_point`**. The square bracket notation (`[..]`) employed on this last conjunct represents generalized function application over some collection—in this case, the locations making up the domain of the labeling function.

The **`initialization`** clause, which concludes the declaration of the `Spiral` model, ensures that new instances

of the type are initialized with their terminal location referencing the center.

The most notable addition (syntactically speaking) to the specification of interface operations is the addition of *parameter modes*. These are intended to indicate to clients the purpose and use of parameters. Specifically, the **evaluates** mode parameter specifies that an expression is expected. An **updates** parameter mode is affected as stated in the operation's contract. Whereas the output value of an **alters** mode parameter is unspecified (to avoid over specification), the input value of a **replaces** mode parameter is ignored, because it would be overridden. The **restores** mode is a shorthand for ensuring that value is not modified by the operation, and finally the **clears** mode indicates that the parameter is initialized.

Each operation is annotated with an optional **requires** clause and an **ensures** clause. The former is a caller responsibility and it indicates what must be true in order for the operation to be called. The latter is an implementor's responsibility, and it states what will be guaranteed after the call terminates.

For example, the `Lengthen` operation **requires** that the spiral 'has room' for another location, and **ensures** the following:

1. The new (outgoing) terminal location is the successor of the #-denoted *incoming* terminal location (`#S.Trmnl_Loc`).
2. The labeling function `S.Lab` remains the same, except at `#S.Trmnl_Loc`—which now maps to the new, user provided label: `#e`.

### 6.2 Formalizing an implementation

As with the prioritizer, we represent our `Spiral` programmatically as a record containing an array `Labl` that holds the various `Label` objects inserted, a `Length` field that keeps a count of the number of items currently inserted, and a `Curr_Place` that maintains a current-location cursor into the `Labl` array. And while this is indeed one means of representing the spiral, it's worth noting that there are many alternative designs one could choose from, such as a pointer based implementation [12]—as suggested in Figure 2.

Two assertions are used to document implementations that provide type representations to aid programmers and automated verification systems.

The **convention** assertion is a representation invariant on the `Spiral` type. It may be assumed before the code for every (external) operation (except initialization), and must hold at the end of each such code (except finalization). In this case the **convention** is used to assert that the current place and length markers, declared in the record, are within bounds (as these are frequently used to index into the `Labl` array).

The **correspondence** assertion—also referred to as an abstraction function (or relation)—is used to connect the details of our programmatic representation of the `Spiral` to its mathematical model defined in its interface contract. For instance we 'connect' the conceptual (**conc**) notion of a `Curr_Loc` to the programmatic `Curr_Place` one by encoding the relationship between the two in terms of counting spiral successor applications.

With the formal contracts concluded, all that remains is the implementation of each procedure. One aspect of the implementation worthy of note is the use of builtin language features such as the swap operator, **:=:** (designed to mitigate the effects of aliasing [9] and reduce the cost of copying large, generic structures). Use of features such as these, coupled with the brevity and simplicity of the implementation itself makes the above component assembly a compelling target for automated verification efforts.

The interface contracts and component implementations in this paper have been already developed using in development version of the RESOLVE compiler and IDE (built on top of the Jetbrain's platform[8]), screenshots of which appear in Figure 7. While the implementations in this paper have not been verified, a variety of other component implementations have been verified [22,5,27].

## 7 Related work and summary

The work described in this paper falls largely into the existing (and rapidly growing) body of work in the realm of model-based engineering (MBE) and component based model engineering (CBME). The rapid growth of these areas can be attributed to the ever increasing complexity of engineering correct software systems, and the inability of existing software engineering development practices and languages to cope with such complexity [21]. As such, many languages and tools have been developed to create and compose models that allow specifiers to rapidly prototype, scale, transform [8], and formalize [2,17] model-based systems at higher levels of abstraction, thus easing creation, validation, and long term maintenance.

The Architecture Analysis and Design Language (AADL) is an Eclipse-based (OSATE) toolchain [7] and associated language used for developing architectural models capable of capturing both the static structure and dynamic runtime characteristics of large and small, embedded and realtime systems. The language supports modeling at various levels of granularity (system, thread, process, subprogram), throughout the software lifecycle, and from a variety of different viewpoints and perspectives. To give the language a certain

[8] `https://www.jetbrains.com/`

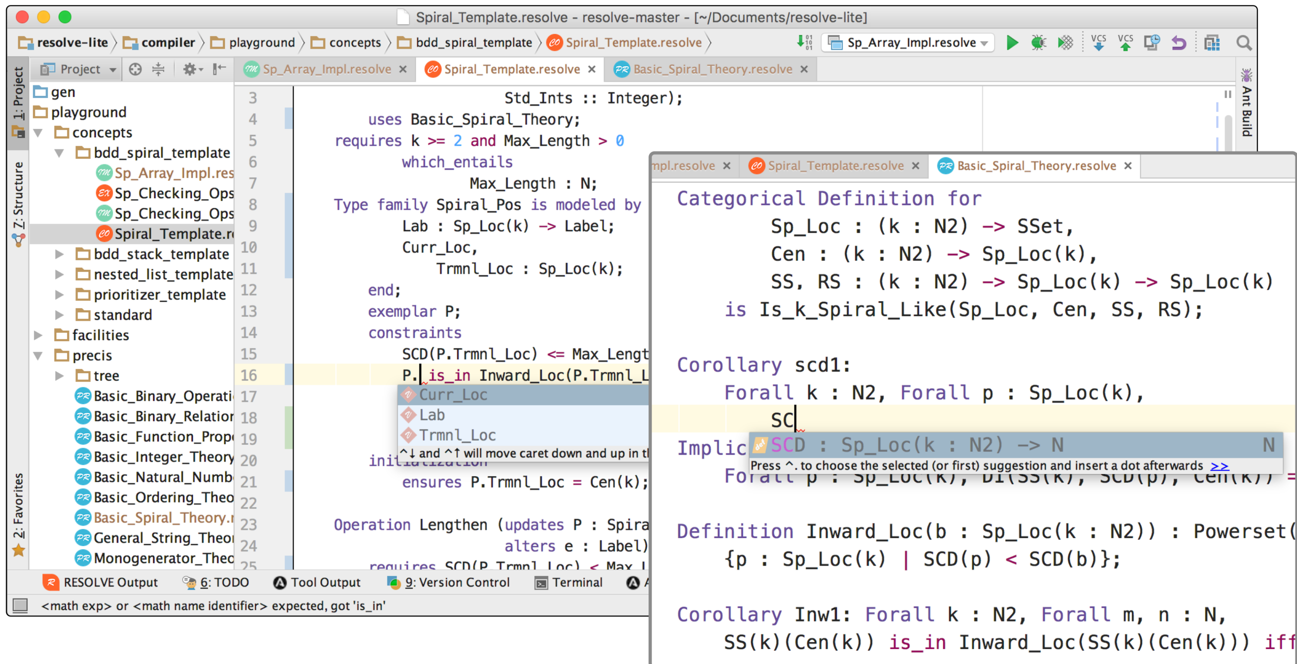


**Fig. 7** A screenshot of `Spiral_Template` (left) and `Basic_Spiral_Theory` (right) using an in development editor for the language built for Jetbrain's IDEs

amount of adaptability and flexibility, the core language is made extensible through the notion 'annex's' which permit additional languages to be layered on top of core AADL code.

While not included in the core AADL syntax, functional contracts and contextual assumptions for interfaces in AADL (ones comparable to sort presented in this paper) can be expressed through annexes containing contracts written in languages such as BLESS (Behavior Language for Embedded Systems with Software) [13] and AGREE (Assume Guarantee Reasoning Environment) [4], among others [19]. Each of these methods employs model checking to ensure component behavior meets the annotations described, and also allows for certain consistency/property checks to take place within the architecture itself (e.g. "all connections have an associated latency").

Other research, focused more on the actual methodology behind producing reliable and predictable software beginning with requirements, such as rCOS [10], rely on the use of models and interface contracts (namely pre-post conditions on ports), as a mechanism for taming complexity in not only mission-critical projects, but also those "complex enough to benefit from usage of tools." To remain widely applicable, rCOS interfaces can be derived and constructed for all phases of development, and (like AADL) from different levels of granularity and for different concerns (e.g. specifying static structure vs. the dynamic behavior). Verification of the requirement models is performed using prototyping tools (e.g. [25]), while liveness and safety properties for actual components are turned over to related model checking tools [3].

In summary, this paper has presented a comprehensive case study that is at the intersection of model-based and component-based software engineering. Using illustrative concrete example applications in the case study, the paper has motivated the development of a variety of reusable concepts, each with its own mathematical modeling to drive the interface contracts and alternative component implementations. While we have indeed developed relevant theories to describe the mathematical models, formal contracts, and multiple component implementations for all concepts discussed in the paper, we have described details for one concept to illustrate the issues in developing models, contracts, and component implementations. While we have automatically verified a number of component implementations, modular verification of the ones presented in this paper remain a work in progress. As such, the theories and concepts will no doubt need to undergo further formalization and development to ultimately facilitate automation.

**Acknowledgements** We thank all members of the RESOLVE Software Research Group (RSRG) at Clemson, Ohio State, and elsewhere. Our special thanks are due to Joan Krone, Bill Ogden, and Bruce Weide for their many insightful suggestions and discussions in formulating the ideas appearing in this paper. This research has been funded in part by US National Science Foundation (NSF) grants CCF-0811748, CCF-1161916, and DUE-1022941.

## References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York, NY, USA (2010)
2. Autili, M., Chilton, C., Inverardi, P., Kwiatkowska, M., Tivoli, M.: Towards a connector algebra. In: Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part II, ISoLA'10, pp. 278–292. Springer-Verlag, Berlin, Heidelberg (2010). URL http://dl.acm.org/citation.cfm?id=1939345.1939377
3. Chen, Z., Morisset, C., Stolz, V.: Fundamentals of Software Engineering: Third IPM International Conference, FSEN 2009, Kish Island, Iran, April 15-17, 2009, Revised Selected Papers, chap. Specification and Validation of Behavioural Protocols in the rCOS Modeler, pp. 387–401. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-11623-0_23. URL http://dx.doi.org/10.1007/978-3-642-11623-0_23
4. Cofer, D., Gacek, A., Miller, S., Whalen, M.W., LaValley, B., Sha, L.: Compositional verification of architectural models. In: Proceedings of the 4th International Conference on NASA Formal Methods, NFM'12, pp. 126–140. Springer-Verlag, Berlin, Heidelberg (2012)
5. Cook, C.T., Harton, H., Smith, H., Sitaraman, M.: Specification engineering and modular verification using a web-integrated verifying compiler. In: Software Engineering (ICSE), 2012 34th International Conference on, pp. 1379–1382 (2012). DOI 10.1109/ICSE.2012.6227243
6. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms, 2nd edn. McGraw-Hill Higher Education (2001)
7. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, 1st edn. Addison-Wesley Professional (2012)
8. Gray, J., Lin, Y., Zhang, J.: Automating change evolution in model-driven engineering. Computer **39**(2), 51–58 (2006). DOI 10.1109/MC.2006.45
9. Harms, D.E., Weide, B.W.: Copying and swapping: Influences on the design of reusable software components. IEEE Trans. Softw. Eng. **17**(5), 424–435 (1991). DOI 10.1109/32.90445. URL http://dx.doi.org/10.1109/32.90445
10. Ke, W., Li, X., Liu, Z., Stolz, V.: rcos: a formal model-driven engineering method for component-based software. Frontiers of Computer Science **6**(1), 17–39 (2012). DOI 10.1007/s11704-012-2901-5. URL http://dx.doi.org/10.1007/s11704-012-2901-5
11. Kruskal, J.B.: On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society **7**(1), 48–50 (1956). URL http://www.jstor.org/stable/2033241
12. Kulczycki, G., Smith, H., Harton, H., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: The location linking concept: A basis for verification of code using pointers. In: Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments, VSTTE'12, pp. 34–49. Springer-Verlag, Berlin, Heidelberg (2012)
13. Larson, B.R., Chalin, P., Hatcliff, J.: NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings, chap. BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software, pp. 276–290. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
14. Leavens, G.T.: Tutorial on jml, the java modeling language. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 573–573. ACM, New York, NY, USA (2007). DOI 10.1145/1321631.1321747. URL http://doi.acm.org/10.1145/1321631.1321747
15. Leavens, G.T.: Jml: Expressive contracts, specification inheritance, and behavioral subtyping. In: Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ '15, pp. 1–1. ACM, New York, NY, USA (2015). DOI 10.1145/2807426.2817926. URL http://doi.acm.org/10.1145/2807426.2817926
16. Mashkoor, A., Yang, F., Jacquot, J.P.: Refinement-based validation of event-b specifications. Software & Systems Modeling pp. 1–20 (2016). DOI 10.1007/s10270-016-0514-4. URL http://dx.doi.org/10.1007/s10270-016-0514-4
17. Merten, M., Howar, F., Steffen, B., Pellicione, P., Tivoli, M.: Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change: 5th International Symposium, ISoLA 2012, Heraklion, Crete, Greece, October 15-18, 2012, Proceedings, Part I, chap. Automated Inference of Models for Black Box Systems Based on Interface Descriptions, pp. 79–96. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). DOI 10.1007/978-3-642-34026-0_7. URL http://dx.doi.org/10.1007/978-3-642-34026-0_7
18. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-order Logic. Springer-Verlag, Berlin, Heidelberg (2002)
19. Olveczky, P.C., Boronat, A., Meseguer, J.: Formal Techniques for Distributed Systems: Joint 12th IFIP WG 6.1 International Conference, FMOODS 2010 and 30th IFIP WG 6.1 International Conference, FORTE 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings, chap. Formal Semantics and Analysis of Behavioral AADL Models in Real-Time Maude, pp. 47–62. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
20. Owre, S., Rushby, J.M., Shankar, N.: Automated Deduction—CADE-11: 11th International Conference on Automated Deduction Saratoga Springs, NY, USA, June 15–18, 1992 Proceedings, chap. PVS: A prototype verification system, pp. 748–752. Springer Berlin Heidelberg, Berlin, Heidelberg (1992). DOI 10.1007/3-540-55602-8_217. URL http://dx.doi.org/10.1007/3-540-55602-8_217
21. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006). DOI 10.1109/MC.2006.58. URL http://dx.doi.org/10.1109/MC.2006.58
22. Sitaraman, M., Adcock, B.M., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H.M., Harton, H.K., Heym, W.D., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.W.: Building a push-button RESOLVE verifier: Progress and challenges. Formal Aspects of Computing **23**(5), 607–626 (2011). DOI 10.1007/s00165-010-0154-3. URL http://dx.doi.org/10.1007/s00165-010-0154-3
23. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Trans. Softw. Eng. **23**(3), 157–170 (1997). DOI 10.1109/32.585503. URL http://dx.doi.org/10.1109/32.585503
24. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
25. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
26. Weide, B.W., Ogden, W.F., Sitaraman, M.: Recasting algorithms to encourage reuse. IEEE Software **11**(5), 80–88 (1994)
27. Welch, D., Cook, C., Sun, Y.S., Sitaraman, M.: A web-integrated verifying compiler for RESOLVE: A research perspective. In: Proceedings of the 7th India Software Engineering Conference, ISEC '14, pp. 12:1–12:6. ACM, New York, NY, USA (2014). DOI 10.1145/2590748.2590760. URL http://doi.acm.org/10.1145/2590748.2590760