Semantically-Rich Composition of Virtual Images

Fábio Oliveira, Tamar Eilam, Michael Kalantar, Florian Rosenberg
IBM T.J. Watson Research Center, Hawthorne, NY, USA
{*fabolive, eilamt, kalantar, rosenberg*}*@us.ibm.com*

***Abstract*—Virtualization promises to reduce data centers' total cost of ownership by enabling the creation of a *small* set of *standardized* building blocks to be shared and used many times in different software stacks. However, without proper methodology and tools, an organization can easily end up with a large number of one-off virtual images, adversely affecting the cost.**

We propose an approach, tool, and algorithms for constructing high-quality, semantically-rich image building blocks that are easy to share, compose, and reuse. In our approach, domain experts codify knowledge of a particular software product (or a combination thereof) in a platform- and cloud-agnostic *software bundle*. Image builders easily construct virtual images by composing a set of standardized bundles. Semantic-based validation guarantees a valid and complete image design. Moreover, we propose algorithms to automate image design by searching for an optimal set of building blocks taking into account multiple metrics such as cost, size, and expected build duration.

I. Introduction

Virtualization, and specifically, virtual image technologies (e.g., VMware [14]) are perceived to hold a tremendous promise to reduce IT management cost. This perception is based on the ability to capture complete software stacks, including multiple software components and configuration, enabling a small number of IT experts to codify IT best-practice patterns in a relatively small set of virtual images to be re-used by an entire organization. The reduction in cost stems from fewer software variations to be maintained.

Nonetheless, multiple roadblocks preclude this desired state. First, constructing a virtual image capturing a multi-component software stack is at least as difficult as the traditional software installation and configuration problem. Second, it is hard to effectively re-use images due to many factors, such as multiplicity of cloud and virtualization platforms, difficulty of finding images based on requirements in a large, non-descriptive repository, and difficulty customizing an image.

Moreover, we observe a worrying phenomenon where enterprises adopting virtualization and automation without a clear methodology for standardization are suffering from an adverse effect on cost — image sprawl [8]. In this situation, the number of virtual images grows exponentially with no re-use, resulting in an increase, rather than a reduction, in management cost. The root cause of this unfortunate phenomenon is that the cost is linked to the number of different components to be maintained, virtual or physical. With no commonality between the virtual images constructed (i.e., lack of standardization), the management cost grows with the number of virtual images; this growth in cost is more significant than the savings due to hardware consolidation.

In this paper, we introduce an approach, architecture, and algorithms to leverage virtualization to achieve standardization, and thus a true reduction in IT management cost. Our proposed approach enables a small number of best-practices experts to construct a relatively small number of *software bundles* codifying best-practice software installation and configurations. Non-expert practitioners can use the set of software bundles to *easily* compose multiple combinations of bundles into virtual images to meet their particular requirements. Bundles and images are self-describing through a formal language and contain built-in customization points. Thus, they are easy to share, re-use, and compose. Our approach is implemented in the IBM Image Construction and Composition Tool [6].

Our models and algorithms allow non-expert practitioners to not only verify compatibility between image building blocks (images and bundles), but also select an optimal bill of materials (base image and bundles) that meets a set of requirements, balancing quality, cost, time, and size. To the latter end, we propose two sets of algorithms (greedy and integer-programming based) that identify the best bill of materials for a given input requirement graph. We implemented and experimented with our algorithms as an extension of [6].

Two principles differentiate our work from the current state of the art: (1) the focus on both software bundles and images as key artifacts to be shared and re-used; and, (2) usage of a formal language based on typed graphs to describe a parameterized and customizable structure comprising a complex set of capabilities, requirements, and automation. The focus on bundles as a key artifact allows an effective collaboration and better re-usability in two aspects: (1) parallelization of the work by IT experts as they codify their knowledge of different software products *independently* in different bundles; and, (2) the same collection of bundles can be used to create virtual images in *different* platforms and clouds. The use of a formal structure allows better sharing of images and bundles and enables sophisticated algorithms to produce close-to-optimal results. In our approach, automation of image composition is based on a rigorous design phase, where compatibility of building blocks is validated and the best bill of materials is identified. Images inherit their formal structure from the bundles used for their construction; thus images can be further re-used as a basis to add new bundles to create new images. This contrasts existing technologies (e.g., CohesiveFT [2]) that require images to be built from scratch every time.

Next, we detail our software-bundle-centric approach to image management (§ II), present our image and bundle models (§ III), describe our algorithms for finding an optimized set of building blocks (§ IV), evaluate our implementation (§ V), summarize related work (§ VI), and conclude (§ VII).

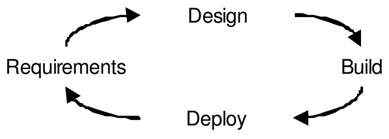


Figure 1. Virtual image life-cycle.

II. Approach to Image Design and Construction

The life-cycle of virtual images is depicted in Figure 1. First, the required image contents are identified. Once the *requirements* are gathered, there is an image *design* phase to select a target virtualization environment (i.e., cloud), and the operating system and software that will be used. Furthermore, deployment time configuration options are identified. Then, the image can be *built* becoming available for *deployment*. As a result of experience with the image, new requirements are gathered for the next iteration of the image development.

This paper focuses on the image design and build phases.

A. Traditional Image Building Approaches

Currently, the most common approach to implementing the image build phase is manual. An *expert* image builder (1) manually creates a virtual machine instance from a base image in the selected virtualization environment; (2) manually installs and configures software on the virtual machine instance; (3) develops and installs deployment time configuration scripts (specific to the target environment) that will run each time the image is deployed to configure/customize the software; (4) captures the virtual machine as a new image; and (5) describes deployment parameters (in a way specific to the target environment), such as network settings and other user preferences, used by the configuration scripts.

This manual approach has several disadvantages. First, it requires the image builder to have significant knowledge and skill both in the target virtualization environment and in the software installation and configuration process. Second, the manual process does not produce any intermediate artifacts that can be re-used when building other images containing the same software. Third, the resulting image has little description of its contents making it hard to find and maintain.

To address these disadvantages, automated image building tools have been developed. Alas, they typically limit software installation mechanisms to a small number, such as RPM [13] [9], forbid deployment time configuration [2] [9], and do not address the problem of identifying image contents, as they rely on poor metadata, preventing proper validation and automation of image design. Furthermore, because of the lack of structure of their images, they do not allow the user to select a pre-built image with non-trivial content to be extended incrementally; rather, they require an image to be always built from scratch.

B. Image Building with Software Bundles

We introduce the concept of *software bundles* and a novel approach to automated image building using them. A software bundle is a *re-usable* asset created by a software expert and can be used to build virtual images in *any* virtualization environment. A software bundle may contain software and one or more image build operations, e.g., scripts to

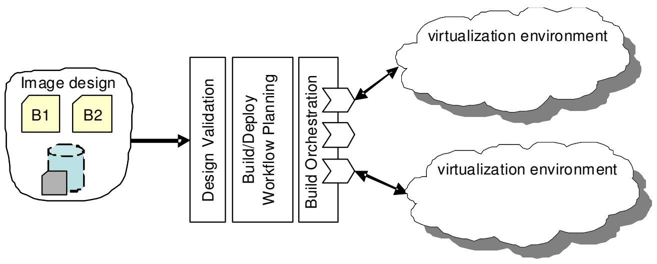


Image design
B1
B2
Design Validation
Build/Deploy Workflow Planning
Build Orchestration
virtualization environment
virtualization environment

Figure 2. Architecture of a virtual image construction tool.

install the software. In addition, it may also contain image deploy operations used to configure the software at image deployment based on user (deployer) preferences and cloud-selected network settings. More importantly, a software bundle includes a formal model describing the software it contains, its requirements for installation and configuration, and a description of the image build and image deploy operations. Software bundles are completely independent of images and virtualization environments.

Using software bundles, a *non-expert* image builder simply defines an image design by selecting a base image to be extended and a set of software bundles that will be added to it. This is done with a virtual image construction tool whose architecture is depicted in Figure 2. Given the design, the tool automatically validates and implements it based on the models of the base image and bundles. This process has three stages. First, the requirements of each software bundle are validated against the capabilities of the image and the other software bundles ensuring the design is valid. Second, a workflow (e.g., a script) is then generated that will execute the image build operations and configure the image deploy operations to run in an order determined by requirement dependencies, operational dependencies, and parameter relationships. Finally, the image is actually built by a general mechanism that deploys the base image, configures and executes the build workflow, and captures the resulting image. This phase relies on plugins for cloud interaction and to generate any cloud-specific artifacts.

The formal models for the building blocks (base image and bundles) are composed into a model for the image design as depicted in Figure 3. The figure represents the selection of (1) a base image that provides the software product DB2 hosted on a Red Hat Enterprise Linux (RHEL) operating system, and (2) a bundle that installs and configures the product WebSphere MQ (WMQ) and requires Linux (any distribution and version). The compatibility between base image and bundle is validated by checking the bundle requirements against the image capabilities: since RHEL is a subtype of Linux, base image and bundle are compatible. Their models are then composed into a new model representing what the new image will provide (DB2 and WMQ hosted on RHEL version 5.6) as well as the deploy operations (and their execution order) of all building blocks used to compose it. Given that the resulting image is self-descriptive, it can be used as a base for building other images using the same process. This incremental image build process obviates the need for constructing an image from scratch every time. Moreover, the models used to describe and validate images and bundles further increase manageability by enabling automated image design, as described in Section IV.

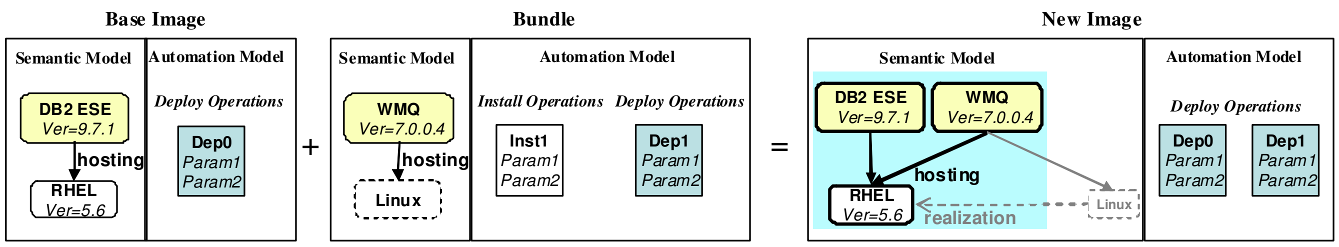


Figure 3. Building images with software bundles.

III. Semantic Models of Building Blocks

This section formally presents the semantic models of image building blocks (see Figure 3). In [5] we describe the automation models for image build and deploy operations.

Our semantic models can be expressed using typed attributed graphs [4] labeled over a pre-defined abstract data type set. Let L_V and L_E be respectively the set of node label and edge label alphabets. Our attributed graph is given by the tuple (V, E, l_V) where $E \subseteq V \times L_E \times V$ and $l_V : V \rightarrow L_V$ is the node labeling function.

We now define some of the core node and edge data types (the sets L_V and L_E) of our modeling language [1]. All node types L_V inherit from a base type termed *unit*. Figure 3 shows an $operatingsystem$ type and a $softwareinstall$ type. All types inherit the Boolean attribute $conceptual$, denoting if the unit must be refined. Other attributes are dependent on the unit type, e.g.: OS distribution, SW product name, and version.

Relationship types L_E include: *hosting*, identifying the runtime container of a component (e.g., DB2 hosted by RHEL); *anti-collocation*, identifying that two units cannot be hosted in a common runtime container; and *realization*, identifying a source conceptual unit ($conceptual = true$) and its target concrete ($conceptual = false$) realizing unit. Concrete units describe the *capabilities* of our building blocks, while conceptual units describe *requirements*. In addition, a conceptual unit may be associated with zero or more $constraints$ on the values of its attributes. When a realization relationship is established between a conceptual unit and a target concrete unit, we say that the concrete unit realizes the conceptual unit. We also say that the requirement expressed by the conceptual unit is satisfied by the concrete unit. In Figure 3 we show the realization relationship between the conceptual *Linux* $operatingsystem$ unit and the concrete RHEL unit. Given that RHEL is a subtype of *Linux* and there are no impeding constraints, RHEL can realize *Linux*. In other words, WMQ's bundle requirement on Linux can be satisfied by the base image. Formally, the semantics of realization is defined as follows. For a concrete target unit U_2 to be a valid *local realization* of a conceptual source unit U_1, U_2's type must be a (non-strict) sub-type of U_1. In addition, values of attributes must "agree", .i.e., they are either the same or satisfy all associated constraints. The realization concept is generalized for multiple units. Specifically, for a *pattern* P containing only conceptual units and a model D, a one-to-one realization function $R : P_V \rightarrow D_V$, where P_V and D_V are the respective sets of nodes in P and D, is termed a *valid realization* of P in D if for every $p \in P_V$, $R(p) \in D_V$ is a valid local realization of p (as defined above), and R defines a subgraph isomorphism for P and D.

Modeling virtual images. As depicted in Figure 3, the semantic model of a virtual image describes the software stack it embodies. The model must contain an instance of an $operatingsystem$ type with attributes to identify the type, distribution, and version of the operating system. Optionally, the semantic model of an image may contain a single instance of the $server$ type identifying the physical server architecture. Further, the model may contain zero or more instances of the $softwareinstall$ type representing the software installed on the image (e.g., DB2). All units of the model must be concrete ($conceptual = false$)—represented by solid rectangles.

Modeling software bundles. The semantic model of a bundle describes the software provided by the bundle as well as its requirements. A bundle includes one conceptual instance of the $operatingsystem$ type, representing the operating system required by the bundle, and zero or more concrete instances of the $softwareinstall$ type, one for each software product that will be installed by the bundle. It may further include zero or more conceptual instances of the $softwareinstall$ type indicating required software products that must be present before the bundle operations can be used. Conceptual units (dashed rectangles) may impose constraints on the values of their attributes, which must be satisfied in any realization.

Composing building blocks. A bundle may be composed with an image if the operating system requirements of the bundle can be satisfied by the image. Software requirements of the bundle may be satisfied by the image, by any bundles already composed with the image, or by bundles composed later. Formally, suppose $I = \{b_1, b_2, \ldots, b_n\}$ is a composition of a *base image* (b_1) and bundles ($\{b_2, \ldots, b_n\}$). Bundle b_{n+1} can be composed with I if the $operatingsystem$ unit in b_{n+1} can be realized by the $operatingsystem$ unit in b_1. Figure 3 shows the composition of a base image and the WMQ bundle.

IV. Image Design as Search

Thus far we have described a process in which a user selects an image design (base image and bundles) and the semantic models are processed to automatically validate the design and build a new image. Our typed graph models described previously enable us to go further and devise an approach to automating and optimizing image design. In this context, we treat image design as a search problem: finding an *optimized* set of building blocks composing an image that *best* satisfies the user requirements. We describe this approach next.

Conceptually, this image search mechanism takes two key inputs: (1) a repository of building blocks (images and soft-

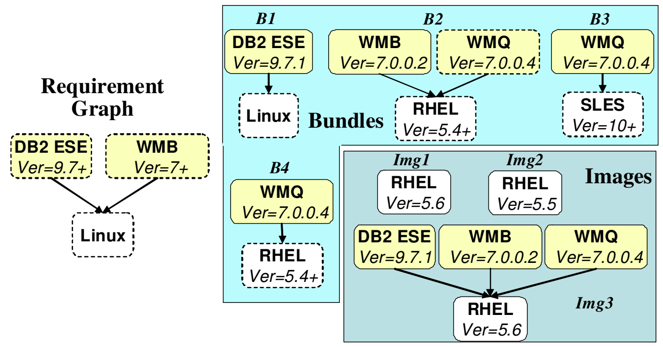


Figure 4. Requirement graph and semantic models of 7 building blocks.

ware bundles); and (2) a *requirement graph*, which is a typed graph represented in the same formal language we use to model software bundles and images. This graph captures the image's desired characteristics, such as operating system features and required software products and version constraints. Figure 4, for instance, shows a requirement graph expressing the need for an image containing DB2 version 9.7 or higher and WebSphere Message Broker (WMB) version 7 or higher, both products hosted on Linux (any distribution and version).

The output of the search is a list of possible solutions $S = [I_1, I_2, ..., I_n]$ such that each solution I_i is a set of building blocks that contains exactly one *base image* and zero or more software bundles. Each solution $I_i \in S$ leaves no requirement unsatisfied, i.e., not only does I_i satisfy all requirements of the requirement graph, but it also satisfies all requirements that may have been added by its building blocks. Note that a solution with no software bundles corresponds to an image that already exists in the repository, whereas a solution containing bundles represents a new image to be built.

Producing a solution I_i entails matching individual semantic models using typed graph isomorphism with constraint checking, the basis of the realization concept (Section III). Given the requirement graph and the collection of images and bundles in Figure 4, the possible solutions would be: $I_1 = \{$Img3$\}$, $I_2 = \{$Img1, B1, B2, B4$\}$, and $I_3 = \{$Img2, B1, B2, B4$\}$. Note that bundle B2 satisfies the requirement on WMB but introduces a requirement on WMQ; that is why bundle B4, which satisfies the requirement on WMQ, is part of solutions I_2 and I_3.

This search mechanism needs to cope with two inter-related challenges in the face of a possibly large repository of images and bundles. First, since there could be many possible solutions, as our simple example illustrates, the problem of ranking them becomes evident. Second, to overcome the combinatorial explosion, it is desirable that the *most important* solutions are found early on in the search process, obviating the need for traversing a large search space in its entirety.

Given this state-of-affairs, we contend that the search process must take into account a metric (or a combination thereof) that translates into the relative importance the user would assign to solutions. Depending on the user goals and needs, different metrics may apply, for example: number of building blocks, number of SW products, licensing costs, expected image build time, and rating of building blocks.

In the sequel, we present two methods for finding the set (or sets) of building blocks that *best* fulfill the user needs.

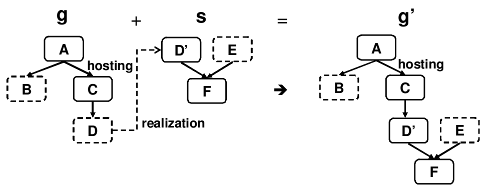


Figure 5. Example of the apply operation.

A. Greedily Guided Search Approach

As suggested above, the naive approach of traversing the entire search space looking for solutions and then sorting them by a ranking metric is not tractable. Rather, we need an algorithm that can find *high-ranking* sets of building blocks while minimizing search space traversal. To that end, we propose that the search be greedily guided by a user-supplied score function so as to reach the *best* solutions (from the user's perspective) first. Our algorithm takes the following inputs: a requirement graph g_0; a set B of building blocks $\{b_1, b_2, ..., b_n\}$; a score function $score(b, g)$, where $b \in B$, g is a requirement graph, and $score(b, g)$ assigns a score b^s to b based on the metrics defined by the user; and, the maximum number of solutions to be found, denoted by max.

In addition, we define the following four operations. The operation $semantic(b)$ takes a building block b as input and returns its semantic model. The Boolean-valued function $satisfied(g)$ returns `true` if the requirement graph g has no requirements left to be satisfied (i.e., no unrealized *conceptual* units—see Section III), or returns `false` otherwise. The operation $apply(g, s)$ takes a requirement graph g and a building block's semantic model s to produce a new requirement graph $g\prime$ that is the result of *realizing* the *pattern* represented by the conceptual units of g with the model represented by s. If s contains additional requirements (conceptual units), they are realized with any concrete units already in g when possible; unsatisfied requirements in s are added to $g\prime$. Figure 5 illustrates the apply operation. Also, note that if the input requirement graph g has a conceptual operating system unit (which means that no image has been applied to it) and s is the semantic model of a software bundle, the operation deals with two conceptual operating system units such that either one can be a subtype of the other. In this case, the operation ensures that the realization involving these units is performed in the right direction. Finally, we define $candidates(g, B)$, which is a building block filtering function that takes a requirement graph g and a set of building blocks B and produces a set $C \subseteq B$ of building blocks such that (1) $\forall b \in C$, b is *compatible* (see below for definition) with g, and (2) $\forall b \in C$, $semantic(b)$ realizes at least one requirement (i.e., conceptual unit) in g with a concrete unit. The second condition guarantees that each candidate building block b reduces the number of requirements present in the *input* requirement graph g.

A building block b is said to be *compatible* with a requirement graph g if (1) the operating system unit of g is realizable by the operating system unit of $semantic(b)$ (or vice-versa), (2) $apply(g, semantic(b))$ does not violate any anti-collocation constraints (see Section III) present in g, and

```
greedily_guided_search($g_0$, B, score, max)
  $S \leftarrow []$;    $I \leftarrow \emptyset$
  recursive_search($g_0$, $I$)

recursive_search(g, $I$)
 if satisfied(g)        // Are all requirements satisfied?
    $S \leftarrow S + [I]$         // Add $I$ to the list of solutions
    return              // Go back one level up in the search tree
 $C \leftarrow$ candidates(g, B)
 if ($C = \emptyset$) return    // If reached a dead end, backtrack; go one level up
 for every $b_k \in C$
    calculate $b_k^s$ = score($b_k$, g)
 create a sorted list $L = [b_1, b_2, ..., b_m]$ such that
    $(m = |C|) \wedge (b_k^s \geq b_{k+1}^s; 1 \leq k \leq m-1)$
 for every $b_k \in L$   $(k \leftarrow 1, 2, ..., m)$
    $I \leftarrow I \cup \{b_k\}$                // Add $b_k$ to a tentative solution
    g′ ← apply(g, semantic($b_k$))
    recursive_search(g′, $I$)        // Descend one level in the search tree
    if ($|S| = max$) return          // If found max solutions, stop search
    $I \leftarrow I - \{b_k\}$
```

Figure 6. Greedily guided search algorithm.

(3) all concrete $softwareinstall$ units (considering attribute values) of $semantic(b)$ are not already present in g. This definition of compatibility guarantees that all software bundles of a solution are compatible with one another and with the operating system of the solution's base image. Moreover, it ensures that a solution contains only unique software products.

Based on the definitions above, we formally present our greedily guided search algorithm in Figure 6. The algorithm returns a list S of solutions that satisfy the requirement graph g_0. Recall that each solution is denoted by a set containing one base image and zero or more software bundles. The order in which the solutions appear in the list S is dictated by the $score$ function provided by the user, which determines how the search space is traversed.

At each search step, the algorithm computes all candidate building blocks based on the current state of the requirement graph, sorts them based on the $score$ function, and iterates over the sorted list of candidates, adding one at a time to a tentative solution. After a candidate is added to a tentative solution, the state of the requirement graph is updated and the same steps are recursively performed on the updated requirement graph. In other words, the algorithm greedily chooses the next step of the search according to the $score$ function, going progressively deeper into the search tree. Upon reaching a leaf, the algorithm either finds a solution or encounters a dead end; in either case, it backtracks and continues until max solutions are found.

In Section V, we evaluate our implementation of this algorithm with a score function aimed to prioritize solutions with fewer building blocks and fewer extra SW products.

B. Binary Integer Programming Approach

In our greedily guided search approach, a score function is supposed to capture the aspects of a solution that are important to the user. The algorithm traverses the search space based on the values assigned by the function to the building blocks at each search step so that the most relevant solutions are found first. This approach is useful when a single score function can express all facets in which the user is interested.

Although sometimes a score function can weigh metrics of interest in a meaningful way, it cannot easily express tradeoffs among multiple (possibly unrelated) metrics. For example, a user may want to maximize the overall user rating of a solution while keeping the licensing costs within a certain limit.

In light of this type of scenario, we propose to model search as a binary integer programming problem. Integer programming problems [11] are special cases of linear programming problems, which are defined as follows. Given a linear objective function of some decision variables, the problem is to assign values to the decision variables such that the objective function is maximized (or minimized) subject to a number of feasibility constraints. The objective function and feasibility constraints are linear functions of the decision variables. For binary integer programming, all decision variables are integers in the domain $\{0, 1\}$. This formulation lends itself to model search scenarios of the type exemplified above.

In what follows, we formulate the problem of searching for an image design as binary integer programming.

Inputs. The inputs to the problem are: (1) a requirement graph R denoting the software and operating system characteristics the user wants; (2) a set B of building blocks $\{b_1, b_2, ..., b_n\}$; (3) an objective function o capturing metric(s) that the user wants to maximize or minimize; and (4) a list F of constraints representing restrictions on certain metrics. The input set B is produced by filtering the repository of building blocks as follows. Each $b_i \in B$ must be *compatible* with the requirement graph R (see definition of compatibility in Section IV-A). Moreover, for each $softwareinstall$ unit $q_j \in semantic(b_i)$, if q_j's type is the same as that of a unit $w_k \in R$, then one of the following conditions must be true: q_j realizes w_k (or vice-versa); or q_j and w_k are identical (same attribute values).

Output. The output is a set $I \subset B$ of building blocks. If the search is satisfiable, I will contain exactly one base image and zero or more software bundles; otherwise, $I = \emptyset$.

Decision variables. We have n binary decision variables, one per building block in B. Let m be the number of software bundles and k be the number of base images, such that $n = m + k$. Let us represent our decision variables by a vector $\overrightarrow{X} = [x_1, x_2, ..., x_m, x_{m+1}, x_{m+2}, ..., x_{m+k}]$ where $x_1 - x_m$ denote bundles and $x_{m+1} - x_{m+k}$ denote base images. A solution reflects an assignment of values to each decision variable. If $x_i = 1$, then $b_i \in I$; if $x_i = 0$, then $b_i \notin I$.

Single-image constraint. In order to guarantee solution correctness, we need to ensure that the assignment of values to the vector $\overrightarrow{X}$ made by the integer programming solver corresponds to a set I that contains exactly one base image. We do so by introducing the following feasibility constraint: $x_{m+1} + x_{m+2} + ... + x_{m+k} = 1$.

Coverage constraints. Recall that the requirement graph R and the semantic models of building blocks represent software and software requirements. To ensure that every requirement (conceptual unit) in R and those introduced by selected building blocks are satisfied (realized by a concrete unit) in the solution, we introduce a set of *coverage constraints*. We formulate them in terms of *coverage vectors* as follows.

Let $\mathcal{U}$ be the set of all units (vertices) of R and of $semantic(b_i)$, $\forall b_i \in B$, combined. For example, in Figure 4, $\mathcal{U}$ contains 18 units. Let $O \subset \mathcal{U}$ be the set of $operatingsystem$ units. Let us define the set of *unique unit sets*, U, to be a set where each element u_i represents a set of units such that $u_i \subset (\mathcal{U} - \mathcal{O})$, $\bigcup_i u_i = (\mathcal{U} - \mathcal{O})$, and $\bigcap_i u_i = \emptyset$. Each set u_i is constructed in such a way that: (1) all of its units are in the same type hierarchy; (2) it may contain identical software units (same type and attributes), conceptual or concrete; and (3) all of its conceptual units are realizable by at least one of its concrete units.

Based on this definition, the set U for Figure 4 contains 3 elements corresponding to the following unique unit sets: u_1 = {DB2 ESE (9.7+), DB2 ESE (9.7.1)}, u_2 = {WMB (7+), WMB (7.0.0.2)}, and u_3 = {WMQ (7.0.0.4)}. For simplicity we show identical units (those with same type and attribute values) only once in each set. Finally, let $\overrightarrow{U} = [u_1, u_2, ..., u_p]$ be a vector created from U that assigns an (arbitrary, but fixed) order to the elements of U.

We now define the coverage vector for R as $\overrightarrow{C^R} = [c_1^R, c_2^R, ..., c_p^R]$ and coverage vectors for each building block $b_i \in B$ as $\overrightarrow{C^i} = [c_1^i, c_2^i, ..., c_p^i]$ where

$$c_j^R = \begin{cases} 0 & \text{if } R \text{ does not contain a unit of the } u_j \text{ set} \\ |B| & \text{if } R \text{ contains a concrete unit of the } u_j \text{ set} \\ -1 & \text{if } R \text{ contains a conceptual unit of the } u_j \text{ set} \end{cases}$$

Similarly, we define values for c_j^i, for each vector $\overrightarrow{C^i}$.

For example, consider the models in Figure 4 and its vector of unique unit sets $\overrightarrow{U}$ as shown above. The coverage vector for the requirement graph would be $\overrightarrow{C^R} = [-1, -1, 0]$, while bundle B2 would have its coverage vector $\overrightarrow{C^{B2}} = [0, 7, -1]$.

We use coverage vectors to define feasibility constraints whose aim is to ensure that the integer programming solver assigns values to $\overrightarrow{X}$ corresponding to a set of building blocks that leave no requirement unsatisfied. For each $u_j \in U$ $(1 \leq j \leq p)$, we define the following feasibility constraint: $(\sum_{i=1}^{n} x_i c_j^i) + c_j^R \geq 0$. The rationale of these constraints is the following. Based on the definition of coverage vectors, each requirement for a unit of u_j corresponds to -1, whereas each time a unit of u_j is provided corresponds to $|B|$. By enforcing the summation above to be no less than 0, we guarantee that if there exists a requirement for a u_j unit, a valid solution needs to have a building block x_i that provides it. That condition holds true even for a solution where $n-1$ $(|B|-1)$ building blocks require a u_j unit and one building block provides it.

Software-uniqueness constraints. To ensure the solution contains unique software products, we need to define another set of feasibility constraints. To that end, we first define a *software vector* $\overrightarrow{S^R}$, associated with R, and n software vectors $\overrightarrow{S^i}$ $(1 \leq i \leq n)$, associated with each $b_i \in B$. Values are assigned to each element s_j of a software vector as follows: if a concrete unit of u_j appears in the model, $s_j = 1$; otherwise, $s_j = 0$. We then dictate the following constraint for each $u_j \in U$ $(1 \leq j \leq p)$: $(\sum_{i=1}^{n} x_i s_j^i) + s_j^R \leq 1$.

Anti-collocation constraints. It is also paramount to guarantee that the solution contains no incompatible software. In our graph models, software incompatibility is represented by anti-collocation links (see Section III). In our integer programming formulation we translate anti-collocation links into the *anti-collocation vectors* $\overrightarrow{A^R}$ and $\overrightarrow{A^i}$ $(1 \leq i \leq n)$. If an incompatibility with a set u_j is indicated in the model, $a_j = 1$; otherwise, $a_j = 0$. Then, we define the following anti-collocation constraint for each $u_j \in U$ $(1 \leq j \leq p)$ and each $b_z \in B$ $(1 \leq z \leq n)$: $(\sum_{i=1}^{n} x_i s_j^i)_{i \neq z} + a_j^z + s_j^R \leq 1$.

OS-incompatibility constraints. Consider the example of Figure 4. Clearly, bundles B2 and B3 cannot be part of the same solution, since they require different distributions of Linux: the former requires RHEL, whereas the latter, SLES. In our integer programming model, we capture that in anti-collocation vectors similar to the ones defined above, and rely on analogous feasibility constraints.

User-defined constraints and objective function. All constraints above are added automatically by processing the graph models to ensure correctness. A set F of constraints may be provided by the user so that certain desired characteristics are enforced. For example, suppose the following quantities are associated with each building block $b_i \in B$: licensing cost (vector $\overrightarrow{L}$) and completion time (vector $\overrightarrow{T}$). For a bundle, completion time is the time it takes for its provided SW to be installed; for an image, the time it takes for it to be instantiated. Then, the user could determine that a valid solution must have the total licensing cost bound by $cost$, i.e., $\sum_{i=1}^{n} x_i l_i \leq cost$. Finally, the user may require a solution with minimum completion time, expressed as the objective function $MIN(\sum_{i=1}^{n} x_i t_i)$. This objective function and all constraints can then be given to an integer programming solver that will produce a solution minimizing the objective function subject to all constraints.

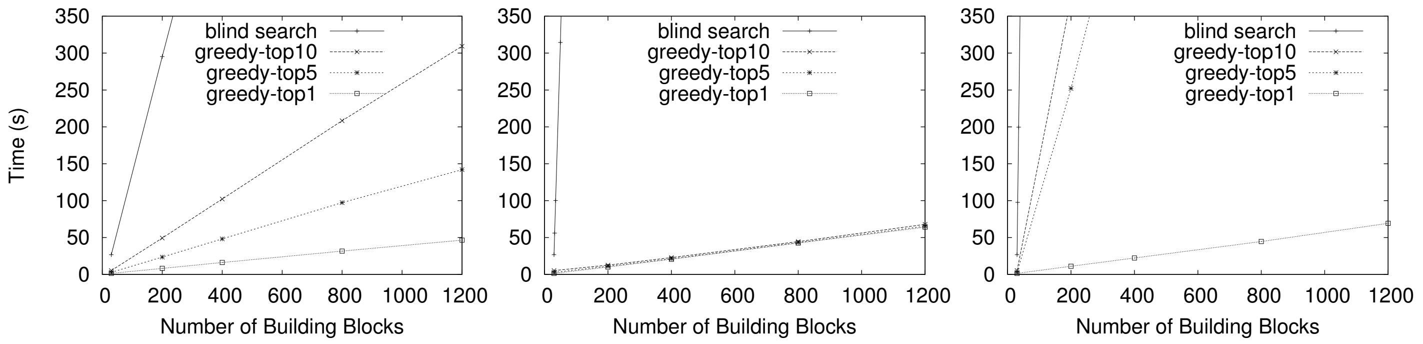
V. Implementation and Evaluation

Our approach to virtual image design and construction (Section II) has been implemented as part of the IBM Image Construction and Composition Tool [6]. We then extended the tool implementing our ideas to identify optimized set(s) of building blocks for automated image design (Section IV). This section highlights key aspects of our implementation and analyzes its behavior under different scenarios.

Greedily guided search. Recall that our algorithm depicted in Figure 6 relies on a score function to prioritize the various options presented by the search space in every step of the way, so that the most desirable image designs (solutions) are found early on. A solution is found when there is no requirement left to be covered in the requirement graph g. In our implementation we propose a $score(b, g)$ function that assigns a score b^s to a candidate building block b as follows: $b^s = reqSat(b, g) - reqAdded(b, g) - swAdded(b, g)$

The functions $reqSat$, $reqAdded$, and $swAdded$ determine how b would change the requirement graph g if b were added to a tentative solution. They compute, respectively, how many: pending requirements are covered (conceptual units realized); new requirements (conceptual units) are added; and, new SW products (concrete units) are added. Based on this heuristic, a candidate is rewarded for each requirement it covers, and penalized for each SW (either required or provided) it adds. By choosing a candidate with the greatest value of b^s (lines 9–18 of Figure 6) the algorithm tries to cover as many requirements as possible in one step (favoring solutions with fewer building blocks), while limiting the number of added SW.

The rationale of this heuristic is twofold. First, a solution with as few building blocks as possible takes advantage of base



(a) – "Diverse repository" (b) – "Multiple solutions" (c) – "Multiple dead ends"

Figure 7. *Run time of the greedily guided search algorithm with 3 different repository constitutions.*

images and bundles that encompass multiple SW, leveraging all testing that has been done by the building block author regarding the interaction of its SW components. Second, the less extra SW in the new image, the smaller its footprint, and the lower its deployment/build time and licensing costs.

Experimental setup. To understand the performance characteristics of our algorithm, we analyze it by running it against repositories of building blocks with varying size and constitution. Figure 7 summarizes the results of these experiments for 3 repository constitutions with sizes ranging from 30 to 1200 building blocks, where 30 building blocks (7 base images and 23 bundles) are common for the 3 cases. For each scenario we then add a different mix of building blocks. In all cases we use the same input requirement graph expressing a search for a Linux image containing 3 SW products: DB2 Enterprise Server Edition 9.7, WMB (WebSphere Message Broker) 7.0.0.2, and WMQ (WebSphere MQ) 7.0.1.4.

Each data point in Figure 7's graphs represents the average of 3 runs, during which we observed negligible variations in the measurements. Each chart shows 4 curves: the one labeled "blind search" represents the run time of an algorithm that blindly traverses the entire search space enumerating all possible solutions, whereas the curves labeled "greedy-top10," "greedy-top5," and "greedy-top1" correspond to different executions of our algorithm for different values of the parameter max (10, 5, and 1, respectively), which determines the maximum number of solutions wanted. The order in which the solutions are found is dictated by the score function.

All experiments were conducted on a blade equipped with 4 3.16 GHz Intel Xeon CPUs and 8 GB of memory, running the 64-bit SLES (SuSE Linux Enterprise Server) 10.2 OS.

Next, we present the results for the 3 scenarios of Figure 7.

Diverse repository. For this case, we add to the initial 30 building blocks a collection of bundles for different SW products extracted from IBM's internal product database. The new bundles satisfy no requirement of the requirement graph.

Figure 7(a) shows that the run time of our algorithm increases linearly with the repository size. Since the bundles added to the repository do not satisfy any input requirement, the set of candidates remains constant, i.e., no new search space paths are added. Therefore, the search time is dominated by the computation of the $candidates$ list (line 9 of Figure 6), which entails iterating over the repository to match each building block's semantic model against the current requirement graph. Since the $candidates$ list is constant, so is its sorting time (lines 13–14). The constant factor of each curve reflects the number of times the $candidates$ list is computed.

Multiple solutions. In this case, we add to the initial 30 building blocks multiple copies of a WMB bundle. When going from 30 to 200 building blocks, we add 170 copies; when going from 200 to 400, we add another 200 copies, and so on. In this repository mix, each m added bundles lead to $6m$ new solutions. Hence, for a repository with 1200 building blocks, the total number of solutions is $12 + 6 \times (1200 - 30)$, as the 30 initial building blocks provide 12 solutions.

Figure 7(b) shows that our algorithm exhibits an $O(n \log n)$ run-time behavior in this case. Furthermore, finding 10 solutions takes approximately the same time as finding 5 and 1, as the three coinciding curves (top-10, top-5, and top1) attest.

Due to our score function, finding the top solution, which comprises a DB2 image and one bundle containing both WMB and WMQ, does not require visiting all extra bundles added to the repository. (Note that this would be true even if the combined "WMB+WMQ" bundle were the one replicated.) Moreover, finding up to 10 solutions does not require visiting more than 9 copies of the WMB bundle. Therefore, the search time is dominated by sorting the list of candidates—$O(n \log n)$—, and in each of these executions (top-10, top-5, and top-1) a $candidates$ list is computed and sorted three times, once for each level of the search tree until a leaf is reached. *The algorithm exhibits this performance characteristic even when we increase the total number of solutions exponentially, e.g., by adding copies of the DB2, WMB, and WMQ bundles simultaneously.*

Multiple dead ends. This repository mix contains building blocks that cause our algorithm to backtrack often due to paths that lead to dead ends. To create such a mix, we first create a WMB bundle with a SW requirement that cannot be satisfied by any building block in the repository; then, we add multiple copies of this bundle as before. Although all added bundles expand the search space, they contribute no new solutions.

Figure 7(c) shows the results for this case. The shape of the "top-1" curve is the same as that of the "top-1" curve of the previous scenario (Figure 7(b)). Differently, however, although the top-5 and top-10 curves also reveal an $O(n \log n)$ run time, their constant factor is higher because the $candidates$

list is computed and sorted more times in these cases.

Our score function allows the algorithm to find the top solution before reaching any dead end. When searching further, however, it backtracks all the way to the first level of the search tree, 2 times for top-5 and 4 times for top-10, totalizing $2m$ and $4m$ dead ends caused by the m extra bundles.

Discussion. The three scenarios described above represent extreme cases of the spectrum of repositories the algorithm may encounter. In all cases, we experimentally showed that, owing to our score function, the top-1 solution could be found quickly, even for large repositories. For a 1200-building-block repository, the top solution was found in 46, 64, and 67 seconds in the first, second, and third scenario, respectively.

The time complexity of the algorithm is $O(kn \log n)$, where n is the number of building blocks and k is the number of times the $candidates$ list needs to be computed and sorted. We expect k to be rather small, specially for the top-1 solution.

When our algorithm needs to backtrack to the top level of the tree, the relative rank of a few solutions may not be produced according to the intent captured by the score function. However, even when desirable solutions are suppressed, equivalent or better ones are presented to the user. Moreover, the top solution is always the *best* (based on the score function), and it is often the only one the users care to see. More importantly, all solutions returned, irrespective of the repository constitution, are always valid and complete, due to our validation of building blocks' models.

Integer programming approach. We have validated our integer programming methodology using the library LPSolve [7]. Due to space constraints, we show no experimental analysis.

VI. Related Work

Sapuntzakis et al. [10] introduce a language to describe *virtual appliances* and its configurable parameters to facilitate reusability. Sun et al. [12] propose an architecture for modeling and deploying complex software topologies. Differently, Dastjerdi et al. [3] leverage ontologies to describe virtual appliances to allow users to select the "best" ones based on Quality of Service. Unlike these efforts, we focus on image design and construction, and our approach to image design automation searches for the best design that satisfies user's requirements while optimizing for desired metrics.

Numerous tools for virtual image construction have been developed, such as, SUSE Studio [13], CohesiveFT's Elastic Server [2], and rPath [9]. These tools rely on a predefined, small number of software installation mechanisms (e.g., RPMs), limit image customizability (e.g., cannot handle deployment parameters), and do not support incremental image construction, requiring images to be built from scratch every time. Worse yet, they rely on poor metadata, preventing them from properly validating and automating image design. On the other hand, our software-bundle-centric approach to image design and construction supports a generic mechanism for bundle installation and allows for full image customizability. Moreover, our semantically-rich models provide the foundations for incremental image build where new images can be built simply by adding more software bundles to them. Finally, our models allow us not only to effectively address the problem of validating an image design, but also to automate and optimize a design by searching on large repositories of images and bundles. Current solutions can only automate image build, but not design. We address both problems.

VII. Conclusions

We proposed a novel approach for virtual image management that revolves around software bundles and incremental image build. Our approach allows for sharing, reusability, and composability of image building blocks. Our experience with our tool suggests that, given a set of software bundles, the time and effort to create virtual images, as well as the resulting number of images to be maintained, are greatly reduced.

Our building block models and incremental image build approach allowed us to pioneer the idea of treating image design as search: given metrics of interest to the user, search a large repository of images and bundles looking for the most desirable image designs. We proposed two general approaches to tackle this problem that use a supplied combination of metrics to drive the search. We also proposed one such metric aimed to produce "best fit" image designs that satisfy the user requirements with as few building blocks and SW products as possible. Finally, we evaluated our greedily-guided algorithm with this metric showing that it can find the top solution quickly even for large building block repositories.

References

[1] W. Arnold et al. Pattern based SOA deployment. In *ICSOC*, volume 4749 of *LNCS*. Springer, 2007.

[2] CohesiveFT. http://cohesiveft.com/.

[3] Amir Vahid Dastjerdi et al. An Effective Architecture for Automated Appliance Management System Applying Ontology-Based Cloud Discovery. In *CCGrid*, May 2010.

[4] Hartmut Ehrig et al. Fundamental Theory for Typed Attributed Graphs and Graph Transformation based on Adhesive HLR Categories. *Fundam. Inf.*, 74:31–61, October 2006.

[5] T. Eilam, M. Elder, A.V. Konstantinou, and E. Snible. Pattern-based composite application deployment. In *IM*, 2011.

[6] IBM. IBM Image Construction and Composition Tool. http://www.ibm.com/developerworks/mydeveloperworks/blogs/complete_clouddev/entry/icct_cloud_tool?lang=en.

[7] LPSolve. http://sourceforge.net/projects/lpsolve/.

[8] Darrell Reimer et al. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *VEE*, 2008.

[9] rPath. http://rpath.com/.

[10] Constantine Sapuntzakis et al. Virtual appliances for deploying and maintaining software. In *LISA*, October 2003.

[11] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[12] Changhua Sun et al. Simplifying Service Deployment with Virtual Appliances. In *SCC*, July 2008.

[13] SUSE. http://susestudio.com/.

[14] VMware. http://vmware.com/.