2010

Formal Reasoning about Software Object Translations

Berzins, Vladis; Luqi; Musial, Peter M.

Formal Reasoning about Software Object Translations*

Vladis Berzins[1], Luqi[1], and Peter M. Musial[1,2]

[1] Computer Science Department, Naval Postgraduate School,
1411 Cunningham Rd., Monterey, CA, 93943, USA
[2] Department of Computer Science, University of Puerto Rico, Rio Piedras,
P.O. Box 23328, San Juan, PR 00931, USA

Abstract. In this work we examine the problem of verifying translations from outputs of one system to the inputs of another system, which we refer to as the *output-to-input translation problem*. We present a formalization of this problem along with a verification mechanism based on constraint logic programming. Composition of systems is an important issue in the software reuse domain, and has applicability in other areas of software engineering such as transformation of information from one phase of the development process to another. Some challenges are to verify the translation mechanisms that may be needed to connect independently designed components and assess to what degree is the consumer component functionality enabled after the composition takes place. To this end we use constraint logic programming modeling techniques. Our model allows formalization of the translation problem in the form of constraints and relations between the outputs and the inputs of involved components. Since CLP tools are computationally expensive, we identify characteristics of translation problems for which our technique is practical. We conclude with an application of our translation framework within the Documentation Driven Software Development methodology.

1 Introduction

In this work we address a problem originating from the domain of software component reuse in the design of complex systems, where a system may be composed of complex subsystems. To reduce implementation costs and improve reliability, design of such systems may incorporate existing software libraries or complete subsystems. In practice software reuse is not limited to just matching component interfaces, and will potentially require translation of outputs of the producer component to the inputs of the consumer component. The challenge is that the translation can potentially involve merging outputs via some sequence of operations. In this case it is important to verify that the translation enables functionality of the consumer software component while supporting the outputs of the source software component.

Therefore, the context for this work is conceptual verification of certifiable systems where there exists a prior version of the system that has been certified correct and one part of it is being updated and the system constraints are known. For example, it may be required to replace some component due to new performance demands or since it

* This work is sponsored through AOR grant for development of the DDD framework. In addition P.M. Musial is funded through the NRC Postdoctoral Fellowship.

C. Choppy and O. Sokolsky (Eds.): Monterey Workshop 2008, LNCS 6028, pp. 43–58, 2010.

became obsolete and unsupported. The update must be verified correct in order for the next version to obtain certification as well. The verification of the updated system does not eliminate the need for testing as the replaced component may contain implementation errors, such as memory overflows, timing faults, etc. The goal of this work is to provide a sound verification framework that validates design of the upgrade, under the assumption that the new component meets its specifications, as expressed by the constraints associated with the component. The verification of the translation targets architecture level faults and should help reduce system integration problems.

1.1 Validating Component Compositions

The problem of validating the composition of components is an active research area. A previously studied approach to the problem is to use an object-oriented modeling language and graphical tools to define and reason about component composition. For example, GenVoca generators [1] are used to synthesize software systems by composing components from reuse libraries. In GenVoca, components are parametrized program transformations that encapsulate consistent data and operation refinements. In [2] it is shown that GenVoca can be used to validate composition of components. However, the assumption in [2] is that the compatible components implement same abstract interface. In our work we do not make this assumption. Moreover, the degree to which the consumer component is enabled as a result of the composition is not being measured in [2], which we measure in this work by computing the range of the output of the consumer component as compared to the output produced before composition takes place. Meaning, constraint logic program can provide information about the possible solution space for the constraint program that defines the new composition based on the set of given inputs, which can be compared to the range of outputs produced for the same set of inputs by the consumer component either on its own or in an existing system configuration.

Another way to approach the problem is through reasoning about interface matching and validation based on the detailed knowledge of the component's code, as was done in [3]. Specifically, [3] presents an approach to modeling components and component composition, which incorporates the notion of communication between individual components in the composition. Verification of composition is performed on interactions between component interfaces. Again, translation is not considered. An interesting aspect of [3] is that the proposed model can check if a specific (transition) path in the composition is reachable. However, the extent of reachable paths is not computed.

In [4] a conceptual framework is presented for software component definition, validation, and composition. This framework is dubbed ComDeValCo and it approaches the problem of component composition by structuring system components as a library of components. The verification and validation methods for composition are not well detailed.

The problem of translation in the context of web applications is examined in [5], where a formal model is presented for providing a mapping between two independent web modules. The key contribution of [5] is that they remove the lexicographical mapping requirements between components and allow the designer to choose which outputs and inputs should be connected. In addition, [5] presents a mechanism for code generation from the mapping. The translations are mathematically specified through abstract

state machines. However, the mapping is a direct mapping (one-to-one) and authors of [5] do not provide a verification mechanism for the mapping. In our work we relax the one-to-one mapping requirement and allow components in the composition to be connected via an intermediate translator.

The above works approach the problem of component composition from a systems level. There are higher level approaches to component composition that involve reasoning at a more abstract level of automata such as Timed Input/Output Automata [6]. However, it is often the case that an abstract automata representation of components may be difficult and time consuming to extract from existing implementations. In our approach we attempt to provide a bridge between the abstract and low level system specifications, where our framework will attempt to verify component translation based on the amount of information given. Clearly, the more detailed information is provided about the components, domain of inputs, range of outputs, and the translation functionality, the more informative the answers will be.

Type-checking is another classic approach that can be used to verify matching between outputs and inputs in the composition. However, static type-checking is not sufficient to verify values of the data based on examination of the code, for example checking bounds on array indexes. Dynamic type-checking can be used to verify variable assignments, but such an approach is usually used at runtime and to the best of our knowledge cannot always be used as a proactive mechanism [7].

In this work we abstract the problem of component composition to its functional behavior. Specifically, we relax assumptions that component code is available, rather we assume that only constraints on the behavior (functionality) of the components is known along with domain information of the input and outputs. These constraints characterize the slots in the architecture that are to be filled by the components in question. The constraints represent the standards imposed by the architecture on "plug-compatible" components. These standards typically do not completely characterize all the details of the behavior of acceptable component, although ideally they should be strong enough to guarantee that any component that meets the standards will enable the architecture to perform its intended functions.

Clearly the more information about components is available the more precise guarantees can be provided about the composition. We do not assume that there will be a strict interface matching between components and that the use of a non-trivial translation layer may be necessary. We are interested in verification of the composition by measuring the degree of enabled functionality of the consumer component after the composition takes place.

1.2 Constraint Logic Programming

Constraint logic programming (CLP) is a programming paradigm that allows expressing logical relations among several unknown variables, where each variable accepts values from some domain. For example (from [8]), assume placement of a square and a circle in a two dimensional plane, where the relation between these two objects is that the circle should be contained within the square. Note that the size of these objects and the location of the circle within the square are not specified. One may add additional constraints to this system that describe specific ratios, distances from the borders, or

add additional objects and introduce constraints on the relations between all objects. A CLP solver is a tool that provides an answer whether the constraints expressed by a CLP program are satisfiable, unsatisfiable, or undecidable.

CLP has been used to successfully model complex problems from various domains, such as design of analog and digital circuits, civil and mechanical engineering, finance, assembly line optimization, building visual language parsers, and many others (for a comprehensive survey of CLP models we refer the reader to [9]).

Solving a CLP program involves the problem of constraint satisfaction [10], which can be a computationally difficult problem. This means that for certain classes of problems constraint satisfaction requires exponential time with respect to the number of variables, for example all problems that are reducible to the 3SAT problem. The good news is that many practical problems can be defined in terms of constraints over finite domains (including some problems from the boolean domain). Programs that require infinite domains are efficiently solvable if constraints can be specified as linear constraints on integer variables; the same is not true for nonlinear constraints. Continuous domains are common in real-world problems, where for this class of problems there are efficient solutions for constraint satisfaction when constraints are expressed as linear inequalities forming a convex region – linear programming. For a brilliant presentation of the constraint satisfaction problem we direct the interested reader to [10]. For practical pointers on modeling decisions that make a solution to a constraint logic program terminate quickly we direct the reader to [11]. Advances in the research on CLP solvers and increasing computational power of computers makes CLP an attractive method for solving problems in a wide range of domains, including software engineering. However, in the context of this work we will introduce restrictions to make our approach practical.

In this work we are interested in validating the translation between two software components, but also in assessing size of the solution space. Doing so can give insight about the functionality being enabled of the target software component. When the size of the solution space produced by the consumer component after the composition is same as the size of the solution space prior to the composition, then the composition preserved the functionality of the consumer component, else some functionality has been restricted.

Document Structure. In Section 2 we present a general framework for component translation. At this point we abstract from software components and present our framework in terms of generic components that have inputs, outputs, and functionality. In Section 3 we introduce a general model within which the translator is defined in terms of constraints and relations between outputs and inputs, and we present some necessary conditions for validation of a given translation. Various constraint domains are discussed in Section 4. In Section 5 we examine application of our framework within Documentation Driven Software Development framework, by augmenting an Open Architecture [12]. We conclude with final remarks and point out future research directions in Section 6.

2 Formal Modeling of Component Translations

Figure 1 depicts a composition of two components that is accomplished by use of a translation of outputs of the source component to the inputs of the target component.

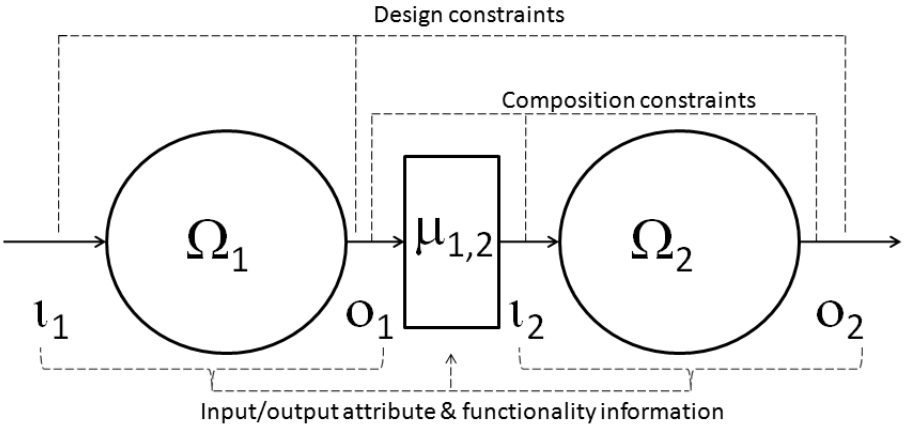

Fig. 1. A general representation of the translation process from outputs of one component to inputs of another

Specifically, outputs of Ω_1 are mapped via a logical relation to the inputs of Ω_2 through the mapping $\mu_{1,2}$. (In the configuration that is being replaced, Ω_2 is connected directly to some other producer component.)

The objective is to verify to what degree the translation $\mu_{1,2}$ enables the functionality of the consumer component. Specifically, the range of output of the consumer component prior to new composition configuration is produced by the functionality of consumer component based on the allowed domain of system inputs. The new composition configuration and the use of a translator may restrict the domain of inputs of the consumer component and consequently reduce the range of its outputs. It may be possible for such differences to be measurable and used as indicators of the degree of enabled functionality.

2.1 Assumptions

The primary assumption is that the translation is unidirectional, where the data flows from the producer component to the consumer component. This is done for presentation purposes and the bidirectional extension is straightforward.

We assume that the components used in the translation are well-formed, with inputs and outputs that are bound to some specific set of types. The well-formedness assumption restricts components to ones that have a fixed functionality and fixed protocol for interacting with their connections. Note that our assumptions are not very restrictive and do not rule out all models that may exceed descriptive powers of the specific CLP language used or may exceed capabilities of the accompanying solver. We provide a short survey of CLP languages and their capabilities and implementations in Section 4.

2.2 Notation and Model

In this section we describe our model along with definitions of mathematical notation and symbols used to describe components and their functionality, as well as the meaning of

the translation. In order to avoid notational clutter, whenever we refer to a component, or component's inputs or outputs outside of the composition we forgo the use of subscripts.

A component Ω is represented as a tuple from $\iota \times o \times R \times C$. The elements of ι and o are tuples with at least one field, more formally $o = \langle o_1, o_2, \ldots, o_n \rangle$ and $\iota = \langle \iota_1, \iota_2, \ldots, \iota_m \rangle$, where n and m represent the number of outputs and inputs respectively. Each field of ι, respectively o, is defined over a domain of some fixed type. R is a mapping from inputs to outputs, such as $R : \iota \overset{\{r_1, r_2, \ldots, r_n\}}{\rightarrow} o$, where $\{r_1, r_2, \ldots, r_n\}$ represents the set of relations that map component inputs to each of the outputs. Components with internal states are modeled by adding an extra input and output for each state variable, so that state variables can be used in constraints. R is modeled as a relation to accommodate possibly nondeterministic component behavior. C is a set of constraints defined over inputs and outputs. Constraints are terms that define desired properties of the inputs and outputs. There should be at least one constraint on each field of ι and o. For instance, assume that the first field of input ι accepts any number from the domain of natural numbers, in this case the minimal constraint on the first field of the input is: $\text{domain}(\iota(1)) = \mathbb{N}$. In addition, C contains constraints that define relations between individual inputs and outputs that realize the component transformation. For example, a component may have two outputs where one is a checksum of the other, hence a constraint would have to be added that describes this relation. Note that $\text{range}(o(\cdot))$ can be defined directly as a set as it was done for $\text{domain}(\iota(1))$, or it can be defined as a set computed over the relation r_i for some i between 1 and n. Specifically: $\text{range}(o(i)) = \{y : \exists x_1, \ldots, x_m (\forall (1 \leq j \leq m \Rightarrow x_j \in \text{domain}(\iota(j))) \wedge y \in r_i(\langle x_1, \ldots, x_m \rangle))\}$, where $1 \leq i \leq n$.

A translation $\mu_{1,2}$ is a mapping from the outputs of Ω_1 to the inputs of Ω_2, and is represented by a tuple of the form $\iota \times o \times R$, where fields of this tuple are defined similarly as above. In addition, $\text{range}(o_\mu(i))$ is defined similarly to $\text{range}(o(i))$ (see the preceding paragraph).

Therefore, given $\mu_{1,2}$, the definitions of Ω_1 and Ω_2, and the set of system constraints, verification of the translation requires that we check at least the following goal conditions.

$$\text{range}(o_1(i)) \subseteq \text{domain}(\iota_{\mu_{1,2}}(i)), 1 \leq i \leq n_{\mu_{1,2}} \tag{1}$$

$$\text{range}(o_{\mu_{1,2}}(j)) \subseteq \text{domain}(\iota_2(j)), 1 \leq j \leq m_2 \tag{2}$$

The above conditions reflect the requirement that the domain of the mapped outputs of Ω_1 via some combination of operators to the inputs of Ω_2 falls into the acceptable domain. Hence, ensuring that the output produced by the mapping of outputs of Ω_1 will be supported by inputs of Ω_2.

The goals can also include assertions that describe the intended cooperation between the components. For a simple example consider a secure internet connection that encrypts data while it is in transit. In such a case component Ω_1 should act as an encoder, component Ω_2 a decoder, and an appropriate additional goal would be that the two components compute transformations that are inverse of each other, which can be expressed as the assertion $o_1 = \iota_2$.

3 Translation Verification

Thus far we presented a general framework that defines the translation from outputs of the producer component to the inputs of the consumer component. The remaining activity is the verification of the translation. This section presents a method for verification of μ by using a *constraint logic programming* [9] framework. The discussion that follows presents an approach for a general, theoretical CLP solver, and not for any specific CLP implementation. However, this general approach is used as a guideline for implementation of CLP solvers.

A constraint logic program can be defined in terms of tuple $\langle G, S\rangle$, where G is a set of constraint goals, and S is a set of constraints. In our model, a minimal set of constraint goals is represented by conditions (1) and (2) in the preceding section. These goals ensure that the communication between the producer component and the translation, and the translation and between the consumer component allow unrestricted information flow. Therefore, our aim is to prove each of the terms in G to be true. The set S represents the set of conditions under which the set of goals has to hold. Therefore, constraints in S are assumed to be true and represent system requirements that have been proved correct beforehand. The solver verifies constraints from G by moving these that evaluate to true to the set S. A system is satisfiable when all constraints in G evaluate to true and the set G is empty at the end of the prove; otherwise, the system is unsatisfiable. Therefore, the solver checks whether all constraints in $G(X)$ and $S(X)$ hold true for all interpretations of system variables in the set X, where X represents the set of system variables. In a well formed goal, the only free variables correspond to the inputs and outputs of the components and the translation itself. Per our assumption each field of each input and output is associated with at least one constraint. Hence all of the free variables that appear in G also appear in S. When the solver is successful the system $\langle G, S\rangle$ is satisfiable, and since G has no free variables other than those in X, the goal assertions must be true for any scenario in which the constraints hold. We conclude that within the composition the ranges of outputs are subsets of domains of the inputs – as desired. (Recall that a constraint system is undecidable when the termination condition cannot be reached, see [13] for debugging techniques.)

Hence, using our model, we convert conditions (1) and (2) to the following set of goals:

$$G = \bigcup_{1\le i\le n_{\mu_{1,2}}} \{\text{range}(o_1(i)) \subseteq \text{domain}(\iota_{\mu_{1,2}}(i))\} \cup \bigcup_{1\le j\le m_2} \{\text{range}(\mu_{1,2}(j)) \subseteq \text{domain}(\iota_2(j))\} \tag{3}$$

The set S is simply:

$$S = C_1 \cup C_2 \cup C_{composition\ constraints} \cup C_{design\ constraints} \tag{4}$$

Where C_1 and C_2 denote the constraints associated with Ω_1 and Ω_2, respectively. $C_{composition\ constraints}$ and $C_{design\ constraints}$ (see Figure 1) represents any additional constraints on the composition of the two objects. For example, the system design may impose that the outputs of Ω_2 have a specific relation to outputs of Ω_1 or inputs

of Ω_2. Therefore, these constraints enforce the semantics of the transformation implemented by the composition of the two components. In practice, the developer does not need to be specific about which constraints belong to G and which belong to S. Rather, the developer in addition to the constraints depicted in (3) identifies all system constraints and the logical relation between the system variables.

The following a very simple example illustrates a situation in which a constraint model allows us to verify semantics of the translation that goes beyond simple type checking. A producer component produces an integer value as an output, and the consumer component accepts an integer value and an input – these requirements represent constraints C_1 and C_2. However, the values output by producer are opposite of what is expected by the consumer component – an example of a composition constraint, $C_{composition\ constraints}$ (i.e., translation implements the following relation $o_1(i) = -1 \times \iota_2(j)$ for some appropriate indexes i and j). The goal constraints are trivial in this example and basically require that outputs and inputs are constrained to the integer domain. Such condition is easy to verify using a constraint model, but may be difficult to verify using other methods.

Assessing The Solution Space. The CLP program as defined by (3) and (4) will produce a set of terms. If the solution set to our program is an empty set, meaning the solution space is empty, then the given translation is too limiting. Otherwise, the degree to which the functionality of Ω_2 is enabled can be assessed by comparing the size of the produced solution space to the size of the domain of ι_2. Note that the range of o_2 represents the solution space produced by Ω_2 in isolation from the new composition configuration, where all values in the range of o_2 were produced as a result of some function of Ω_2. CLP solvers support verbosity levels that allow printout of the reachable solutions and the decisions made along the way. The solution trace will depend on the characteristics of the program and the CLP language/solver that is being used. This means that the composition can be re-evaluated against the system design requirements and validated against the trace printout of the reasoning leading to possible solutions. Trace analysis could be performed by a system designer or possibly be automated. Such analysis can uncover whether the conceptual output of the system is correct or not. If the output is not correct, then the analysis can help to identify which constraints or functionality of Ω_1 or $\mu_{1,2}$ are responsible for unnecessarily restricting the desired behaviors of the newly composed system. (For additional information on debugging constraint programs please see [13].)

4 Extent of Our Approach

CLP Scheme [14,9] defines classes of languages denoted as CLP($\mathcal{X}$), where $\mathcal{X}$ is a pre-interpretation defining the primitive constraints, functions, and their interpretation. Specifically, this description contains the following information [15]: *signature*, which defines a set of function and predicate symbols and associates arity with each symbol, hence defining the terms and primitive constraints of the constraint language, *domain*, which defines the intended interpretation of the constraints, *theory*, which describes the logical semantics of the constraints, and *solver*, which is a description of a mechanism that can determine where a program described in the language is satisfiable,

unsatisfiable, or undecidable. Next we list the more prominent classes of domain constraints. The following summary is based on the CLP survey [9].

- $\mathcal{R}$: The signature consists of linear arithmetic operators, and constants 0 and 1. The domain is the real number set. Hence CLP($\mathcal{R}$) is a language that supports arithmetic operations over real numbers.
- $\mathcal{R}_{Lin}$: a constraint domain defined similarly to $\mathcal{R}$ with arithmetic operation $*$ removed. Basically, limited to linear inequalities.
- $\mathcal{Q}_{Lin}$: defined similarly to $\mathcal{R}_{Lin}$, but restricted to the domain of rational numbers only.
- $\mathcal{FT}$: the signature contains a collection of constant and function symbols and the predicate $=$. The domain is a set of finite trees. The primitive constraints are equations between terms. Basically, it is the Herbrand constraint domain, i.e. based on equations on the algebra of finite terms.
- $\mathcal{RT}$: defined similarly to $\mathcal{FT}$, where the difference is that the domain includes infinite trees.
- $\mathcal{FEAT}$: the signature consists of a binary predicate symbol $=$, a set of unary predicate symbols (called sorts), and a set of binary predicate symbols (called features). The domain is a set of trees (not necessarily finite), where nodes are sorts and edges are features. Hence, the constraint domain is defined over feature trees.
- $\mathcal{WE}$: is a language is defined over strings and characters with the concatenation and equality operations. Hence, it is a constraint domain of equations of strings.
- $\mathcal{BOOL}$: the signature consists of 0, 1, and operators $\neg$, $\wedge$, $\vee$, $\otimes$, $\Rightarrow$, and $=$. The domain is limited to two values: true and false. Hence, this is a two-valued Boolean constraint domain.
- $\mathcal{FD}$: is a constraint domain that is referred to as the finite domain. The signature contains operators $+$, $=$, $\neq$, and $\leq$, and the domain is restricted to a bounded set of integer values.

The literature contains other constraint domains (see [9]). Tools exist that support the above languages (although not all features may be supported). Some examples of CLP tools include BNR-Prolog [16], CAL [17], CHIP [18], CLP($\mathcal{R}$) [19], Prolog family [20], RISC-CLP(Real) [21], and Choco [8]. Each of these tools supports one or more $\mathcal{X}$ and provides solvers that are able to answer if a CLP program is satisfiable, unsatisfiable, or is undecidable under supported $\mathcal{X}$. The Choco solver is especially interesting since it is a constraint programming system that can be used to define a software architecture for variable domains, constraints, propagation and tree search and implements the basics of a constraint system.

In addition to the above CLP languages, there exist other more exotic CLP related languages such as: REF-ARF [22] which is essentially a procedural language and it supports non-determinism because of constraints used in conditional statements. REF-ARF also supports statements such as $x = x + 1$, where such statements are treated as constraints of the form $x_{i+1} = x_i + 1$. In [23], it is shown how to translate concurrent systems with infinite state spaces to CLP programs while preserving the semantics in terms of transition sequences. Another CLP relative is the Oz [24] language, which contains most of the concepts of the major programming paradigms, including logic, functional (both lazy and eager), imperative, object-oriented, constraint, distributed,

and concurrent programming. Oz has both a simple formal semantics and an efficient implementation – The Mozart Programming System [25]. Oz is a concurrency-oriented language that makes concurrency both easy to use and efficient.

To sum up, there is a rich set of CLP languages that can be used to describe and reason about component compositions. CLP solvers are becoming more powerful and are able to provide solutions to a plethora of practical problems. Choice of the specific tool is based on the nature of problem that needs verification and the functionality of the convolved components.

5 Application Examples

Software systems in civilian and military domain are increasing in complexity and have ever more significant impact on human safety, financial resources, and national security. Complexity of these systems requires incremental development by design of subsystems which are composed together to yield the complete complex system of systems.

Large scale software systems share any subset of the following properties: long development time, global deployment strategies, mission critical requirements, significant resource demands, timing constraints, high quality and reliability standards, ease of reconfigurability, and interoperability with other systems.

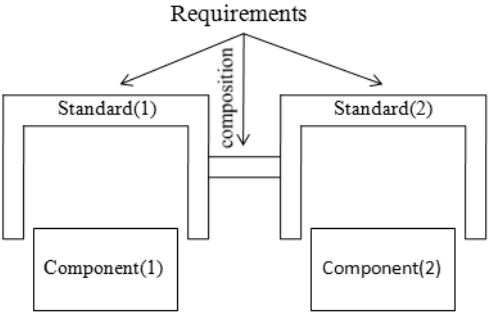

Fig. 2. Open architecture

The key challenges encountered during design of complex systems include: how to generate high quality and high confidence software, how to support system evolution and accommodate changing requirements, how to enable support for variety of stakeholders, and how to improve efficiency and productivity of the development process. The feature that ensures successful development, implementation, deployment, and sustainability is precise documentation.

Documentation Driven Software Development (DDD). The DDD [26] framework is a software engineering methodology that provides assistance for all software life cycle processes, most notably, requirements gathering, quality assurance, design, system evolution and re-engineering, and project management. Each of the software life cycle stages involves communication between stakeholders and the development teams. These two groups share the same objective, but their expertise is in different and sometimes mutually unfamiliar domains. DDD provides mechanisms that allow project information to be effectively communicated between all involved parties, hence providing a bridge between domain of the stakeholders and the domain of developers (which is software design and implementation). Finally, the developers and stakeholders will utilize software and hardware tools during each of the software life cycles. The challenge here is to ensure proper transformation of project requirements, which may be

specified informally, into the formal and mathematical format that is required by the utilized tools. The DDD framework provides mechanisms that help to do just that. Documentation is backbone of the DDD framework. The novel part of the DDD methodology is that all aspects of project information are considered as documentation, which means that documentation is not only limited to natural language text representing system design specifications, manuals, etc. but also includes formal models, knowledge bases, code, simulations, etc. With this definition, the documentation in our approach can provide more effective support for the entire development process.

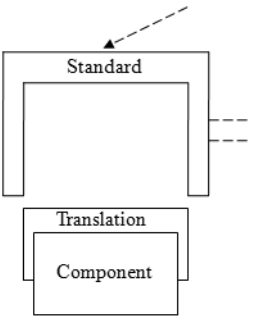

Fig. 3. Component translation wrapper

5.1 Global Architecture

An objective of the DDD framework is to enable systematic construction of reliable software architectures for mission critical systems, particularly with respect to timing constraints extracted from requirements. To this end an open architecture was proposed in [12,27] that allows dynamic system reconfiguration and potentially reduces system testing time. This is achieved by use of standards, requirements/capabilities and environmental assumptions along with components, connections, and constraints (see Figure 2).

As depicted in Figure 2, standards are developed from system requirements. Composition of subsystems is performed on standards. This approach allows system to be tested by examination of standards and their interactions. Implementation of the standard is represented as a plug-in component. Components can be replaced at any time, and the testing needs to be localized to the component/standard interaction. However, authors in [12,27] do not provide detailed explanation of how the interaction between the component and the standard is tested. Our framework can be used to reason about composition of the plug-in and the standard.

5.2 Applications in Open Architecture

The open architecture presented in [12] is an ideal candidate to apply our translation framework. Specifically, the standards used in that context should be well defined entities where functionality and outputs are well documented. A reasonable assumption is that the plug-in component is not an undocumented black-box and partial or complete information about its inputs, outputs, and functionality is known a priori to the developers. However, since standards are inflexible entities, available legacy the plug-in components may be complex to easily modify and its behaviors may not exactly match that of the standard, translation may be necessary.

Figure 3 depicts under the Open Architecture presented in Figure 2 where we use translation in conjunction with standards and components. Under our assumptions, the designer should have ample information to use our framework and test whether the component supports a sufficient range of outputs in order to enable the functionality of the standard, and whether the translation preserves all system constraints. Moreover, our framework enables verification of compatibility of the new component prior to the

installation, hence avoiding system downtime improving confidence in the system after the new component is installed. Furthermore, by forcing the developer to think about the functionality and the composition in terms of the translation and constraints, the testing process for the new component can become more streamlined, where the test cases can be developed based on CLP program – i.e., test outer bounds of the constraints.

5.3 Enumeration Example

This section presents a simple example that is representative of a large class of problems. Specifically, we consider pattern matching via simple enumeration. This type of problem is often difficult to verify for a human, and is commonly found in genetics, security, and logic circuits.

Consider a system consisting of two connected components. Without loss of generality, we can assume that there are two versions of the system, the old version and the new version that is an updated based on the old system – perhaps the old producer component may need to be replaced due to incomparability with the new deployment platform.

Old system. Component one, Ω_1, has one input and two outputs. Component two, Ω_2 has two inputs (outputs are not important). Component one inputs are from the alphabet $\{1, 2, 3, 4\}$, and component one produce 5 output values from the following sets: output one {AB, BA, CA, CB} and output two {A,B,C}. Note that these are the simple constraints on the domains and ranges, i.e., C_1 from equation (4 in Sec. 3). A system constraint is that the only legal inputs to component two are the following pairs: ⟨ AB,C ⟩, ⟨ BA,C ⟩, ⟨ CA,B ⟩, and ⟨ CB,A ⟩ – constraints making up C_2. The meaning of the listed tuples is ⟨ input value to $\Omega_2\iota(1)$, input value to $\Omega_2\iota(2)$ ⟩ . We know that in the old system the component one adheres to the system constraint and implements the following relation: R_{old}: {⟨ 1,⟨ AB,C ⟩ ⟩, ⟨ 2,⟨ BA,C ⟩ ⟩, ⟨ 3,⟨ CA,B ⟩ ⟩, ⟨ 4,⟨ CB,A ⟩ ⟩} – a constraint in $C_{composition\ constraints}$.

New system. We are asked to replace the component Ω_1 with a new component, $\Omega_{1(new)}$, where the replacement has the following specifications: The new component: one input over the alphabet $\{1, 2, 3, 4\}$; and three outputs, two of these are over the alphabet $\{A, B, C\}$, and one over alphabet $\{A, B\}$. Specification for the new replacement states that R_{new}: {⟨ 1,⟨ C,A,B ⟩ ⟩, ⟨ 2,⟨ C,B,A ⟩ ⟩, ⟨ 3,⟨ B,C,A ⟩ ⟩, ⟨ 4,⟨ A,C,B ⟩ ⟩} – this constraint represents $C_{1(new)}$.

The engineer is told that concatenation of some combination of the two outputs should produce the output sequences consistent with the old component. Therefore, the translation element is reduced to a simple string concatenation problem. Where the inputs of to the translation are defined as the outputs of Ω_{new}, and its outputs are defined as inputs of Ω_2. The goal set G in equation (3) defines constraints on domains and ranges of inputs and outputs between Ω_{new} and $\mu_{new,2}$, and $\mu_{new,2}$ and Ω_2. The question is whether this can be validated to be true. Clearly in this case an engineer can preform the manual computation and answer the question accurately and with a manageable degree of effort. However, the same may not remain true as the logic and patterns to match become more complex.

CLP provides an alternative to the manual process. A simple Choco [8] program, see Figure 4, can be written to test various plausible combinations of the outputs and to

verify whether the new replacement component can be used. The answer to our problem is *yes*, where $\Omega_{1(new)}o(2) \cdot \Omega_{1(new)}o(3) \Rightarrow \Omega_2\iota(1)$ and $\Omega_{1(new)}o(1) \Rightarrow \Omega_2\iota(2)$. The designer chose this connection pattern and the solver checked that it does satisfy all of the intended constraints.

```
import static src.choco.Choco.*;
// Import list is truncated in order to improve presentation.
import src.choco.*;

public class TranVer {
   private static IntegerVariable a = makeEnumIntVar("a", 1, 1);
   private static IntegerVariable b = makeEnumIntVar("b", 2, 2);
   private static IntegerVariable c = makeEnumIntVar("c", 3, 3);

   // Defines relation between the inputs to the system and expected input values to Omega 2
   private static Constraint systemTestOmegaTwoInput(IntegerVariable[] tup) {
      return makeExpression(or(
            and(eq(var(tup[0]),var(1)), eq(var(tup[1]),var(a)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(c))),
            and(eq(var(tup[0]),var(2)), eq(var(tup[1]),var(b)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(c))),
            and(eq(var(tup[0]),var(3)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(b))),
            and(eq(var(tup[0]),var(4)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(a)))));   }

   // Defines relation between the inputs to the system and expected input values to Omega 1
   private static Constraint systemTestOmegaOneOutput(IntegerVariable[] tup) {
      return makeExpression(or(
            and(eq(var(tup[0]),var(1)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(a)), eq(var(tup[3]),var(b))),
            and(eq(var(tup[0]),var(2)), eq(var(tup[1]),var(c)), eq(var(tup[2]),var(b)), eq(var(tup[3]),var(a))),
            and(eq(var(tup[0]),var(3)), eq(var(tup[1]),var(b)), eq(var(tup[2]),var(c)), eq(var(tup[3]),var(a))),
            and(eq(var(tup[0]),var(4)), eq(var(tup[1]),var(a)), eq(var(tup[2]),var(c)), eq(var(tup[3]),var(b)))));   }

   public static void main(String[] args) {

      Solver pb = new CPSolver();
      IntegerVariable omega1in1 =   makeEnumIntVar("Omega1.in(1)", 1, 4); // input to Omega 1
      IntegerVariable omega1out1 = makeEnumIntVar("Omega1.out(1)", a.getBinf(), c.getBinf()); // output 1 of Omega 1
      IntegerVariable omega1out2 = makeEnumIntVar("Omega1.out(2)", a.getBinf(), c.getBinf()); // output 2 of Omega 1
      IntegerVariable omega1out3 = makeEnumIntVar("Omega1.out(3)", a.getBinf(), b.getBinf()); // output 3 of Omega 1
      IntegerVariable omega2in1b1 = makeEnumIntVar("Omega2.in(1).b1", a.getBinf(), c.getBinf()); // input 1 bit 1 of Omega 2
      IntegerVariable omega2in1b2 = makeEnumIntVar("Omega2.in(1).b2", a.getBinf(), b.getBinf()); // input 1 bit 2 of Omega 2
      IntegerVariable omega2in2 =   makeEnumIntVar("Omega2.in(2)", a.getBinf(), c.getBinf()); // input 2 of Omega 2

      // Definitions of problem variables
      IntegerVariable[] varsOmega1 = new IntegerVariable[4];
      IntegerVariable[] varsOmega2 = new IntegerVariable[4];
      varsOmega1[0] = omega1in1; // input to Omega 1
      varsOmega1[1] = omega1out1; // output one of Omega 1
      varsOmega1[2] = omega1out2; // output two of Omega 1
      varsOmega1[3] = omega1out3; // output three of Omega 1
      varsOmega2[0] = omega1in1;
      varsOmega2[1] = omega2in1b1; // input one (bit 1) of Omega 2
      varsOmega2[2] = omega2in1b2; // input one (bit 2) of Omega 2
      varsOmega2[3] = omega2in2; // input two of Omega 2

      // Relation implemented by the translation
      Model m = new CPModel();
      m.addConstraint(eq(omega1out2, omega2in1b1));
      m.addConstraint(eq(omega1out3, omega2in1b2));
      m.addConstraint(eq(omega1out1, omega2in2));

      // Constraints defining relations expected inputs of Omega two
      m.addConstraint(systemTestOmegaTwoInput(varsOmega2));
      // Constraints defining relations expected outputs of Omega one
      m.addConstraint(systemTestOmegaOneOutput(varsOmega1));

      pb.read(m);
      CPSolver.setVerbosity(CPSolver.SOLUTION);

      // Invokes the Choco solver
      if (pb.solve()) {
         // Print values of problem variables when solution is reached
         do {
            for(int i = 0; i < varsOmega1.length; i++) {
               System.out.print(varsOmega1[i].pretty());
               System.out.print("_");
            }
            for(int i = 1; i < varsOmega2.length; i++) {
               System.out.print(varsOmega2[i].pretty());
               System.out.print("_");
            }
            System.out.println("_");
         } while (pb.nextSolution() == Boolean.TRUE);
      }

      if (!pb.isFeasible())
         System.err.println("No_solutions_can_be_found._");
   } }
```

Fig. 4. Choco program verifying translation logic from example given in Section 5.3

5.4 Applications in Plug-in Technology

Proposed framework has applications in the military operations where system safety is often critical. Following is a brief description of one possible scenario where system safety requires verification of component composition.

Component translations are needed in applications where the data are sought for a mission support application that performs *electromagnetic spectrum* (EMS) predictions used for surveillance and analysis of radar and communication signals in areas where military forces will operate. This EMS prediction model has relatively stable data requirements that are provided from various data sources. The EMS application presents the prediction results of the model as graphical charts or as graphical overlays on a map. Figure 5 depicts a majority of the data required by the EMS prediction model and the EMS prediction model outputs.

Our approach allows verification of compatibility of plug-ins with the expected standards of the EMS module. The process verifies the data formats of the various data sources against the data formats supported by the EMS model. If new data feeds become available that are not in a standard format (or are in a new standard format), the developers manually reverse engineer the translating code to derive the data required by the EMS application. Similarly, our approach can be applied while handling the outputs of the EMS module. Verification of plug-in compatibility prior to implementation of the needed translations has potential to reduce risk and production cost, and increases confidence in the final system.

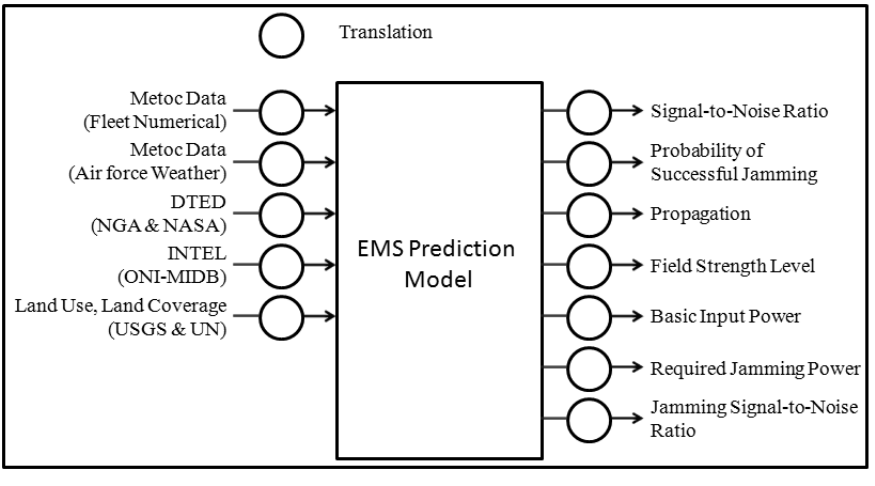

Fig. 5. EMS component and its plug-ins

6 Conclusion

In this paper we present a novel way of modeling and analyzing software compositions that advocates software reuse and increases confidence in the system composition. Although the proposed framework is limited to the types of systems that can be modeled

within the constraint logic programming schema, it can significantly impact reliability and testing processes of systems that are numerically and computationally intensive. Practicality of our approach depends on the ability to express system constraints in terms of the classes of CLP for which constraint satisfaction can be performed in an efficient time. We also describe an application of our framework within the powerful DDD software engineering methodology.

We conclude with some open issues relevant to the presented subject. We are interested in developing methods that allow us to automatically compute attributes describing inputs, outputs, and functionality of the component. In conjunction with advanced natural language techniques for system requirement processing and based on the computed component attributes we are interested in developing methods that are capable of extracting the system of constraints automatically. Component attributes should be computed based on the available documentation, hence requiring natural language processing, and based on the existing black-box (respectively gray-box) reverse engineering methods. Specifically, we are interested in computation of the domain bounds on the inputs and outputs, and collecting a sufficient number of results to verify or estimate component's functionality. Since the above computations are based on imprecise representations of somewhat uncertain information, the proposed infrastructure, needs to be integrated with validation procedures to increase the dependability of the results. Another interesting research direction is automated test scenario generators that use component attributes and functionality, and system constraints (such as domain bounds), where these compute non-trivial testing scenarios to check whether complex component realizations meet the standards associated with a given slot in an open architecture. Resolving these issues will help to reduce the gap between the conceptual verification of upgrade design and the verification of the physical implementation of the new subsystem and strengthen the chain of evidence connecting the original raw data about stakeholder needs to quality assurance procedures for concrete system components..

Acknowledgments. We would like to thank Dagohoy Anunciado from SPAWAR for helpful discussions and insightful comments.

References

1. Gen Voca, `http://www.program-transformation.org/Transform/GenVoca`
2. Pârv, B., Motogna, S., Lazăr, I., Czibula, I., Lazăr, L.: ComDeValCo – a framework for software component definition, validation, and composition. Studia Universitatis Babes-Bolyai Informatica LII(2), 59–68 (2007)
3. Speck, A., Pulvermüller, E., Jerger, M., Franczyk, B.: Component composition validation. International Journal of Applied Mathematics and Computer Science 12(4), 581–589 (2002)
4. Batory, D., Geraci, B.: Composition validation and subjectivity in GenVoca generators. IEEE Transactions on Software Engineering 23(2), 67–82 (1997)
5. Cicchetti, A., Di Ruscio, D.: Decoupling web application concerns through weaving operations. Sci. Comput. Program. 70(1), 62–86 (2008)
6. Lynch, N.: Distributed Algorithms. Morgan Kaufmann Publishers, San Francisco (1996)
7. Meijer, E., Drayton, P.: Static typing where possible, dynamic typing when needed (2005)
8. Choco Constraint Programming System, `http://choco.sourceforge.net`

9. Jaffar, J., Maher, M.: Constraint logic programming: A survey. Journal of Logic Programming 19(20), 503–581 (1994)
10. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. Prentice Hall Series in Artificial Intelligence (2003)
11. Bartak, R.: Theory and practice of constraint propagation. In: Proceedings of the 3rd Workshop on Constraint Programming in Decision and Control (2001)
12. Luqi: Dependable software architecture based on quantifiable compositional model. Technical Report NPS-CS-08-003, NPS (January 2008)
13. Deransart, P., Hermenegildo, M., Maluszynski, J. (eds.): DiSCiPl 1999. LNCS, vol. 1870. Springer, Heidelberg (2000)
14. Jaffar, J., Lassez, J.L.: Constraint logic programming. In: Proceedings of 14th ACM Symposium on Principles of Programming Languages, pp. 111–119 (1987)
15. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
16. Benhamou, F., Older, W.: Programming in CLP(BNR). In: Proceedings of 1'st Workshop on Principles and Practice of Constraint Programming (1993)
17. Aiba, A., Sakai, K., Sato, Y., Hawley, D., Hasegawa, R.: Constraint logic programming language CAL. In: Proceedings of the International Conference on Fifth Generation Computer Systems, pp. 263–276 (1988)
18. Dincbas, M., Van Hentenryc, P., Simons, H., Aggoun, A.: The constraint logic programming lanugage CHIP. In: Proceedings of the 2nd International Conference on Fifth Generation Computer Systems, pp. 249–264 (1988)
19. Jaffar, J., Michaylov, S., Yap, R.: The CLP(R) language and system. ACM Transactions on Programming Languages 14(3), 339–395 (1992)
20. Colmerauer, A.: An introduction to prolog III. Communications of the ACM 33(7), 68–90 (1990)
21. Hong, H.: RISC-CLP(Real): Logic programming with non-linear constraints over the reals. In: Benhamou, F., Colmerauer, A. (eds.) Constraint Logic Programming: Selected Research, pp. 133–159. MIT Press, Cambridge (1993)
22. Fikes, R.: REF-ARF: A system for solving problems stated as procedures. 1, 27–120 (1970)
23. Delzanno, G., Podelski, A.: Constraint-based deductive model checking. International Journal on Software Tools for Technology Transfer 3(3), 250–270 (2001)
24. Van Roy, P., Haridi, S.: Concepts, Techniques, and Models of Computer Programming. MIT Press, Cambridge (2004)
25. Van Roy, P. (ed.): MOZ 2004. LNCS, vol. 3389. Springer, Heidelberg (2005) (Revised Selected and Invited Papers)
26. Luqi, L.Z., Berzins, V., Qiao, Y.: Documentation driven development for complex real-time systems. 30, 936–952 (2004)
27. Berzins, V., Rodríguez, M., Wessman, M.: Putting teeth into open architectures: Infrastructure for reducing the need for retesting, pp. 285–312 (May 2007)