Synthesis of Software from Logical Constraints

Kevin Lano and Shekoufeh Kolahdouz-Rahimi
Dept. of Informatics, King's College London, London, U.K.

Keywords: Model-driven Development, Agile Development, Software Synthesis.

Abstract: This paper presents the case for constraints (requirements formalised as logical assertions) as the key starting point for software development. We describe how system development from such constraints can be automated.

1 INTRODUCTION

Software development methodologies such as Model-driven Architecture (MDA) and Model-driven Development (MDD) have placed increasing emphasis upon the modelling of systems at a level which abstracts from specific platforms and implementation technologies, and even from the details of particular design strategies.

In this paper we describe an MDD approach based on synthesis of executable systems from constraints and declarative models, termed *constraint-driven development* (Lano, 2008).

2 CONSTRAINT-BASED SPECIFICATION

A unifying concept across many kinds of software system is the notion of *constraint preservation* (or *invariant preservation*): the system purpose can be characterised as being to maintain the truth of some properties relating real-world elements (people, artifacts, devices) and their representations within the system, and to maintain properties that inter-relate these representations.

For example:

1. A reactive controller for a lift system must maintain invariants for safety (if a lift is moving, its doors must be closed) and performance (the response time of the lift system to respond to a request should not exceed 5 minutes).
2. An online banking system should maintain data integrity invariants for its persistent data storage (that each account has a valid primary owner, that customers must have valid ages and names, etc), and invariants relating the stored data to the real world elements they represent (each real-world customer has an entry in the stored data which accurately represents their personal attributes such as name and address, etc, and vice-versa).
3. A machine translation system should ensure that the sentences and texts that it outputs (eg., in Russian derived from English input) has the same semantics as the input natural language sentences and text.
4. A software development tool which allows co-construction of UML class diagrams and state machine diagrams should maintain consistency between these models (eg., the transitions in the state machine for a class must be triggered by operations or events of the class).

In each case, we could represent the intention of the system as the maintenance of some set

$$C1, ..., Cn$$

of constraints. Sometimes these constraints are explicit and defined as part of the requirements analysis of the system (eg., cases 1 and 4 above), but more often they remain implicit and are not usually expressed as system requirements (eg., cases 2 and 3).

Constraints and invariants, despite their apparently static nature, are more fundamental than behaviour-based descriptions of a system, because system behaviour can be derived and deduced from the constraints, whilst the reverse is not generally true.

If some event occurs which causes or may cause the violation of a constraint, then the system must react/respond to this event in such a manner as to prevent the violation.

For example:

1. If the lift door sensor indicates that the doors are open, the lift motor actuator should be set off.

Lano K. and Kolahdouz-Rahimi S..
Synthesis of Software from Logical Constraints.
DOI: 10.5220/0004101903550358
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 355-358
ISBN: 978-989-8565-19-8

2. If a new customer is accepted by the online bank, their details should be added correctly to the persistent data store.
3. If a new input text is presented to the translation system, its output text must be a semantically correct translation of the input.
4. If the set of operations of a class are altered on the class diagram editor (eg., by deletion of an operation), then the state machine must change accordingly (eg., all transitions triggered by the deleted operation should be removed).

If the constraints are sufficiently complete and explicit, then the precise action required in response to each potentially constraint-breaking event can be automatically deduced.

3 A SPECIFIC CONSTRAINT-BASED METHODOLOGY: UML-RSDS

UML-RSDS is a model-based development approach with the following principles:

- Systems should be specified at a high level of abstraction, to promote reuse. The abstract specifications are *computation-independent models* (CIMs) in terms of the MDA, and are based on the system constraints, expressed in UML models such as class diagrams, use cases, state machines and interactions.
- The generation of a design and implementation from the specification should be automated as far as possible, so that correct executable systems can be rapidly developed from abstract specifications, and so that changes to the executable versions of the system can be carried out in an agile manner by modifying the specification and regenerating the code.

The approach is supported by an extensive toolset (http://www.dcs.kcl.ac.uk/staff/kcl/uml2web) which has been applied to a wide range of software systems over 15 years (Lano et al, 2003b; Lano and Kolahdouz-Rahimi, 2011a).

Related tools and methodologies are xUML from Abstract Solutions Ltd. (http://www.kc.com) and Perfect Developer from Escher Technologies (http://www.eschertech.com/products/). Specifications in these approaches however require the explicit definition of operations as actions or pseudocode, which are effectively detailed designs at the platform-independent modelling (PIM) level in MDA terms. UML-RSDS also supports direct design at this level, but in addition it provides a higher level of abstraction at the level of constraints, and this is the recommended style of specification in UML-RSDS.

A simple system specification in UML-RSDS consists of a system data model defined by a UML class diagram and system functionalities defined by use cases, with various constraints, expressed in a simplified form of the UML constraint language, OCL:

1. Class invariants.
2. Global constraints of a class diagram.
3. Operation pre and postconditions.
4. Use case pre and postconditions.
5. State machine state invariants. and transition guards.

For example, we could have a class diagram such as that of Figure 1, with a global constraint defining that the value of x for a particular A object is the sum of the z values of its attached C objects:

$$x = br.cr.z.sum$$

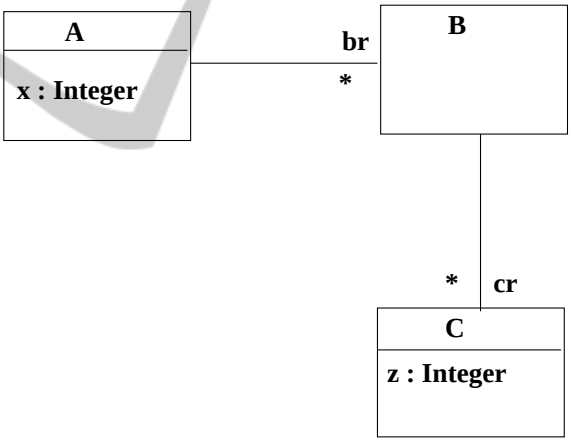

Figure 1: Composition of associations.

Changes to *br*, *cr* and *z* may affect the truth of the constraint, and hence will require some response code to re-establish the constraint. Such response code can be mechanically calculated using the concept of *weakest precondition*. The weakest precondition of a constraint *Cn* with respect to an action or activity *act* is denoted by $[act]Cn$, and is the most general condition prior to execution of *act* which is sufficient to ensure that *Cn* holds after performing *act*.

We consider the following basic incremental changes to a model:

- $setatt(v)$ – Set the value of attribute *att* to *v*
- $setrole(v)$ – Set the value of association end *role* to *v*

- *addrole*(vx) – Add vx to the collection-valued association end *role*
- *removerole*(vx) – Remove vx from the collection-valued association end *role*
- *setrole*(i,vx) – Set the i-th element of a sequence-valued association end *role* to vx.

Table 1 shows examples of wpc calculations for these different kinds of operation, for local class invariants P of the class to which the operation belongs.

Table 1: Local constraint wpc calculations.

Operation act	*Weakest precondition* $[act]P$
setatt(v)	$P[v/att]$
addrole(vx)	$P[(role \cup \{vx\})/role]$
removerole(vx)	$P[(role - \{vx\})/role]$
setrole(i,vx)	$P[(role \oplus \{i \mapsto vx\})/role]$

In the example specification, *br* is a local feature of A, so the calculations of Table 1 apply to give the condition

$$x = v.cr.z.sum$$

which must be established in order that *setbr*(v) re-establishes the constraint.

In other words, the code of *setbr*(v) must perform both its basic action

$$br = v$$

and the additional action

$$x = v.cr.z.sum$$

derived from the constraint. These actions can be performed in parallel or sequentially in either order, as neither action writes to data read by the other.

We can derive similar change-propagation code for *addbr*(vx) and *removebr*(vx).

More interesting, and more challenging, is the case of actions which affect non-local (ie., global) constraints. For this example, changes to *cr* and to *z* are in this category. The response to such changes is performed in a class called *Controller*, which has access to all objects of the system. In this class are operations *setatt*$(obj : C, v : T)$ which apply *setatt*(v) to object *obj*, *addrole*$(obj : C, vx : D)$ which apply *addrole*(vx) to *obj*, and *removerole*$(obj : C, vx : D)$ which apply *removerole*(vx) to *obj*.

Globally, a constraint P based on class A has the meaning that for all instances of A, P holds:

$$A{\rightarrow}forAll(P)$$

Table 2 shows the response code for such global operations, where $A{\rightarrow}forAll(P)$ contains the modified feature f of the operation, and this is not a local feature of the class A upon which the constraint is based.

Table 2: Global operation wpc calculations.

Operation act	*Weakest precondition* $[act]P$
setatt(ob,v)	$A{\rightarrow}forAll(ob \sim self\ implies$ $P[((ref - \{ob\}).att \cup \{v\})/ref.att])$
setrole(ob,v)	$A{\rightarrow}forAll(ob \sim self\ implies$ $P[((ref - \{ob\}).role \cup v)/ref.role])$
addrole(ob,vx)	$A{\rightarrow}forAll(ob \sim self\ implies$ $P[(ref.role \cup \{vx\})/ref.role])$
removerole(ob,vx)	$A{\rightarrow}forAll(ob \sim self\ implies$ $P[((ref - \{ob\}).role \cup$ $(ob.role - \{vx\}))/ref.role])$

In the second, third and fourth cases, *role* is a many-valued association end.

In each case the condition $ob \sim self$ expresses that the object *ob* is reachable via associations of the system from *self*, a specific object of A. *ref* is the navigation route from *self* to *role*, ie, *role* occurs in an expression *ref*.*role* in P. The computation is iterated over all the occurrences of the feature in P.

In the example specification, this results in the following response code in *addcr*$(bx,crxx)$:

$$A{\rightarrow}forAll(bx : br\ implies$$
$$x = (br.cr \cup \{crxx\}).z.sum)$$

This defines a *for* loop over the instances of A. Only the elements of A linked to bx via *br* need to modify their x values to maintain the global constraint: other instances of A cannot be affected by the change to $bx.cr$.

In some cases a fixpoint computation is needed to enforce a constraint. An example is the calculation of the maximum inheritance depth in a class diagram. If generalisations have an integer attribute *depth* to express their depth in the inheritance hierarchy, this feature is characterised by the two constraints:

$$Generalization{\rightarrow}forAll(g \mid$$
$$g.general.generalization.size = 0\ implies$$
$$g.depth = 1)$$
$$Generalization{\rightarrow}forAll(g \mid$$
$$g.general.generalization.size > 0\ implies$$
$$g.depth = 1 +$$
$$g.general.generalization.depth.max)$$

Since *depth* is both read and written in the second constraint, a fixpoint iteration is used to enforce it.

From the computed response codes, complete Java implementations of the operations of individual classes and the controller class can be synthesised. These operations are guaranteed to maintain the invariants (assuming the correctness of the Java virtual machine, the operating system and hardware, etc).

In some cases a constraint *Cn* cannot be used to produce executable response code, but instead it is maintained by checking if application of an operation

op with particular parameter values would break the constraint:

$$[op(pars)]not(Cn)$$

and if so, refusing the request for the operation. That is, $[act]Cn$ is taken as a *precondition* of the operation, where *act* is the basic effect of the operation.

In general, given an operation *op* with basic code *act*, and a constraint *Cn*, the UML-RSDS tools will identify if (i) *op* can invalidate *Cn*, and if so, whether (ii) response code or a precondition should be used to react to or prevent this invalidation.

The first check for a match between an operation and a constraint is:

$$(I): \quad wr(act) \cap rd(Cn) \neq \varnothing$$

ie., some feature or entity updated by *op* is read by *Cn*, or

$$(II): \quad wr(act) \cap wr(Cn) \neq \varnothing$$

ie., both the operation and constraint write to the same entity or feature.

In some cases (eg., *addbr* and a constraint $s \subseteq br$ with $br \notin rd(s)$) the operation cannot invalidate the constraint, despite the apparent data conflict.

If (I) holds and (II) does not, and if *op* definitely affects *Cn*, then if *Cn* has an explicit operational interpretation as an activity $stat(Cn)$, then this activity is added to the code of *op* to restore *Cn*, as described above. If *Cn* does not have an operational form, then $[act]Cn$ is added instead as a precondition of *op*.

If (II) holds and *op* affects *Cn*, then $[act]Cn$ is added as a precondition of *op*.

4 EVALUATION

The UML-RSDS approach has been used for substantial applications, such as the specification and implementation of model slicing tools (Lano and Kolahdouz-Rahimi, 2011b) and migration of UML 1.4 models to UML 2.1 (Lano and Kolahdouz-Rahimi, 2010). The constraint-based specifications are usually more concise than procedural specifications in languages such as Kermeta (Drey et al, 2009), and have been shown to be more comprehensible (Kolahdouz-Rahimi et al, 2012).

REFERENCES

Drey, Z. and Faucher, C. and Fleurey, F. and Mahe, V. and Vojtisek, D., Kermeta Language Reference Manual, https://www.kermeta.org/docs/KerMeta-Manual.pdf, April, 2009.

Kolahdouz-Rahimi, S. and Lano, K. and Pillay, S. and Troya, J. and Gorp, P. V. *Goal-oriented measurement of model transformation methods*, submitted to Science of Computer Programming, 2012.

Lano, K. and Clark, D. and Androutsopoulos, K. *Formal specification and verification of railway systems using UML*, FORMS 2003.

Lano, K. and Clark, D. and Androutsopoulos, K. *RSDS: A subset of UML with precise semantics*, L' Objet, vol. 9, no. 4, pp. 53–73, 2003.

Lano, K. *Constraint-driven development*, Information and Systems Technology, 50, 2008, pp. 406–423.

Lano, K. and Kolahdouz-Rahmi, S. *Model migration transformation specification in UML-RSDS*, TTC 2010.

Lano, K. and Kolahdouz-Rahimi, S. *Model-driven development of model transformations*, ICMT 2011, 2011.

Lano, K. and Kolahdouz-Rahimi, S. *Slicing techniques for UML models*, Journal of Object Technology, vol. 10, pp. 11: 1–49, 2011.