

Checking Program Synthesizer Input/Output

Emanuel S. Grant^{1*}, Jon Whittle², and Rajani Chennamaneni¹

¹ Department of Computer Science
University of North Dakota
Grand Forks, North Dakota, USA

`grante@cs.und.edu`

² QSS Group Inc.
NASA Ames Research Center
Moffett Field, California, USA
`jonathw@email.arc.nasa.gov`

Abstract. The use of program synthesis systems to generate executable code from high level specifications is growing. A fundamental issue faced when using program synthesizers is testing that the synthesized code is a correct implementation of the input specification. Program synthesizers are typically complex artifacts that make use of advanced software engineering techniques such as generative programming and component composition. This makes it difficult to check the relationship between the inputs and outputs. We present an approach to checking the input/output relation of program synthesizers that uses a domain-specific modeling language to specify the expected input and output and constraints between them. We apply our ideas to AUTOFILTER, a program synthesizer for mathematical state estimation problems.

1 Introduction

The goal of program synthesizers is the automatic generation of software for families of applications within a specified domain. Program synthesizers are one of the many approaches to delivering customized software products quickly and cost efficiently from existing libraries of program components and/or parameterized templates and schemas [2, 7]. Program synthesis systems that can generate fully executable code from high level behavioral specifications are rapidly maturing (e.g. [11]), in some cases to the point of commercialization (e.g. SciNapsee [1]).

A fundamental issue faced when developing program synthesis technology is that of checking the correctness of the derived *output code* with respect to the *input specification*. Program synthesizers are typically complex pieces of code. Testing such systems is especially difficult because it can be hard to predict exactly how domain knowledge is instantiated/composed to produce concrete programs — by design, program synthesizers must react to a wide variety of

* This work was partially funded under the RIACS/NASA Ames Summer 2002 Student Research Program.

possibly unanticipated inputs. Program synthesizers may incorporate advanced techniques that are difficult to test with traditional methods – for example, there may be search involved during the code generation process which may lead to a large number of possible paths to verify.

We will present a technique that addresses the problem of checking the input and output of program synthesizers. Regression testing is not enough to validate a domain extension or maintenance step because testing deals only with a pre-defined set of example problems. We propose that the program synthesizer developer create models of the input and output and specify constraints over those models. The input and output of the synthesizer may then be checked for conformance to these constraints, which are embedded in the DSML. Test cases check only a single example, but the input/output constraints are defined at the domain modeling level which means that they are applicable to all examples generated by the program synthesizer. Checking constraints is a lightweight checking method that goes beyond testing but does not involve formal verification.

To model the input/output relation, we propose the use of models that are derived from a *domain-specific modeling languages* (DSMLs). Our DSML is based on the *Unified Modeling Language* (UML) and *Object Constraint Language* (OCL) [4]. We will illustrate the use of these techniques on a specific application – that of checking the input/output relation of AUTOFILTER, a program synthesizer for mathematical state estimation problems.

2 Description of AUTOFILTER

In the following sections, we describe an application of our checking techniques to the AUTOFILTER program synthesizer, a synthesizer for state estimation problems – i.e., problems concerning the estimation of the state of an object (e.g., its position, attitude or noise characteristics) based on noisy sensor measurements. The most common way of solving a state estimation problem is to use a recursive update algorithm known as a Kalman filter [5] which provides a statistically optimal estimate of a state based on a *process model* of the dynamics of the problem under study and a *measurement model* of how the sensor measurements relate to the state.

Given process and measurement models, a Kalman filter can be implemented that optimally estimates the state. At each time step, the new measurement is read in and used to provide an updated state estimate and updated error covariance based on this new measurement information. Because measurements are only available at discrete time steps, the state estimate and covariance is projected ahead to when a new measurement will be available.

AUTOFILTER takes as input a mathematical specification including equations for the process and measurement model but also descriptions of the noise characteristics and filter parameters. From this specification, it generates code that implements (some variant of) a number of standard Kalman filter algorithms. There are many variations of the Kalman filter algorithm, each variation being chosen according to the problem specifics. For example, a nonlinear problem is

usually solved by an extended Kalman filter. Generic Kalman filter algorithms are represented in AUTOFILTER’s knowledge base as uninstantiated *schemas* or programming language independent uninstantiated code fragments. By instantiating and composing fragments, AUTOFILTER can generate code for highly complex filter configurations. AUTOFILTER has been applied to a number of realistic case studies, such as thruster control for automated spacecraft docking and code generation of part of the attitude control system for Deep Space I, a deep space probe.

The Kalman filter domain is a complex one. AUTOFILTER also has a complex schema instantiation and composition mechanism. As a result, AUTOFILTER can generate code in unpredictable ways. This is both a strength and a weakness of AUTOFILTER. It allows interesting solutions to be generated to deep problems but also makes it difficult to keep track of the correctness of the results of code generation. In order to address the latter issue, we applied our input/output checking techniques to AUTOFILTER. The main concern with AUTOFILTER generated code is that code fragments will be composed that are inconsistent with each other. A schema can be thought of as having a number of slots which can be instantiated by the schema itself or calls to other schemas. If two schemas instantiate different slots, however, there is a danger that the slots will be instantiated inconsistently due to a bug in the domain implementation. It is time-consuming to check slot consistency by hand but by developing independent models of how slots should be connected, it is possible to check slot consistency automatically.

3 DSML Development

From the analysis of the family of state estimation problems and Kalman filters a DSML is developed that captures the key concepts of the input/output domains as first-class primitives. The syntax of the DSML is determined from the domain models that result from the domain analysis [9] activity. The semantics is determined from the domain-specific constraints specified during domain analysis. There are four tasks involved in developing DSML, briefly described as (for full details see [3]):

1. *Create Static Concept Stereotypes* A stereotype is a UML extension mechanism that gives additional semantics to a UML model element. This semantics can be specified informally or formally. In our work, stereotypes are specified formally using UML’s OCL, as illustrated in Table 1.
2. *Create Dynamic Concept Stereotypes* Dynamic concepts are stereotyped in a similar way to static concepts. Since our example does not use dynamic concepts, we defer to [3] for a description of dynamic concept stereotypes.
3. *Packaging Stereotypes* For scoping purposes, stereotypes may be packaged into separate profiles. These profiles are then packaged into a “super-profile”.
4. *Create and Package Domain Meta-Models* A meta-model can be defined to describe the relationships between stereotypes in the DSML. These relationships may be existing UML model elements (e.g., aggregation) or may be domain-specific (e.g., specialized UML associations).

The DSML is used to model the input/output of the program synthesizer. Models were developed for AUTOFILTER's using static models of state estimation problems and Kalman filter implementations. The domain models created were UML class diagrams (CDs) that captured the static components of the input and output of AUTOFILTER, and are illustrated in Figures 1 and 2.

Table 1. Class *Process Model* stereotype

Stereotype	process model
Base Class	Class
Parent	N/A
Tag	mandatory
Constraint	<pre> context «process_model» inv: self.isAbstract = true and self.isLeaf = false and self.isRoot = true and self.mandatory = true and self.«state_vector»→size() = 1 and self.«process_noise»→size() = 1 and (self.«control_vector»→size() = 0 or self.«control_vector»→size() = 1) and self.«measurement_model»→size() = 1 and self.«equation».multiplicity = 1 and self.«equation».changeability = frozen and self.«equation».visibility = public and self.«linear».multiplicity = 1 and ... self.«time_variant».multiplicity = 1 and ... self.«value_type».multiplicity = 1 and ... (self.«equation»→forAll(eq eq.initialValue.body = 'discrete') xor self.«equation»→forAll(eq eq.initialValue.body = 'continuous')) and (self.«linear»→forAll(lr lr.initialValue.body = 'true') xor self.«linear»→forAll(lr lr.initialValue.body = 'false')) and (self.«time_variant»→forAll(tv tv.initialValue.body = 'true') xor self.«time_variant»→forAll(tv tv.initialValue.body = 'false')) and (self.«value_type»→forAll(vt vt.initialValue.body = 'absolute') xor self.«value_type»→forAll(vt vt.initialValue.body = 'incremental')) and self.«control_vector»→notEmpty() implies (self.«control_vector».«time_variant»→forAll(tv tv.initialValue.body = 'true') xor self.«control_vector».«time_variant»→forAll(tv tv.initialValue.body = 'false')) </pre>

4 Input/Output Checking

The checking of AUTOFILTER input/output was carried out in a semi-automatic manner with the use of an analysis tool for UML. The *USE* tool is used in verifying the syntactic and semantic constraints expressed in UML models. *USE* is a UML OCL verifier, developed as a PhD research project and available at: www.db.informatik.uni-bremen.de/projects/USE/. A *USE* specification is a description of a UML CD model with OCL constraints, and an object diagram description. *USE* verifies the object diagram description against the CD model and constraints.

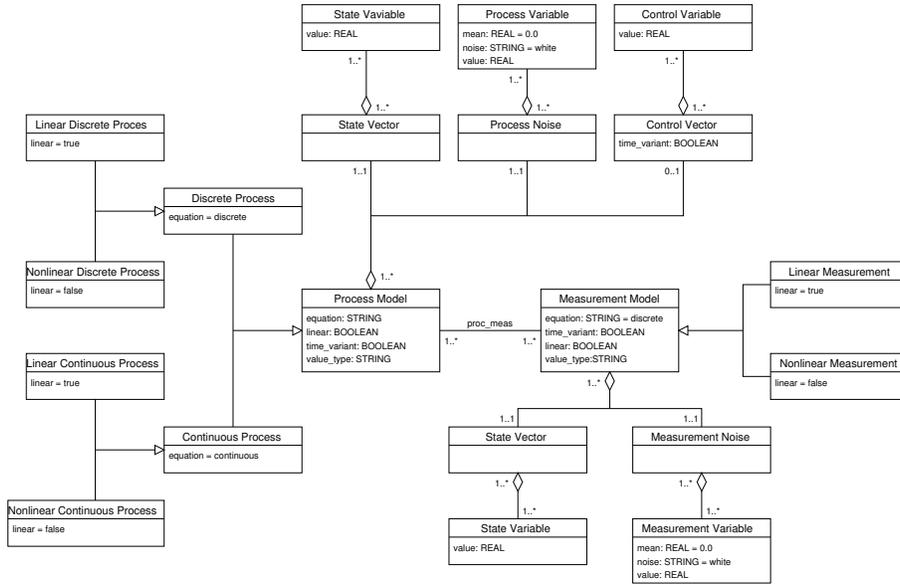


Fig. 1. AUTOFILTER's input CD domain model

In our example, the object diagrams (specifications of instances of AUTOFILTER's input and output) are checked against the domain CDs of Figures 1 and 2. The UML CD and OCL constraints of the DSML stereotypes are converted to *USE* format that are used in the checking of the object diagram specifications. A segment of the *USE*-specific OCL constraints is listed in Table 2. *USE* requires that all constraints be labeled (*KalmanFilter00*, *KalmanFilter30*, *Initialization20*, etc).

Table 2. AUTOFILTER stereotype constraints for *USE*

context Kalman_Filter inv KalmanFilter00 : (self.process_Model.equation = 'discrete' xor self.process_Model.equation = 'continuous')
context Kalman_Filter inv KalmanFilter12 : self.process_Model.control_Vector → notEmpty() xor self.process_Model.control_Vector → isEmpty()
context Kalman_Filter inv KalmanFilter30 : self.process_Model.control_Vector → notEmpty implies self.process_Model.control_Vector.oppositeAssociationEnds() → forAll(oae oae.ordering = #ordered_)
context Initialization inv KalmanInitialization20 : self.imax → forAll(im self.kalman_Filter.declaration.dmax → includes(im))
context Transition_Update inv TransitionUpdate02 : (self.loop.kalman_Filter.measurement_Model → forAll(mm mm.time_variant = true) implies self.measurement_Transition → notEmpty()) xor (self.loop.kalman_Filter.measurement_Model → forAll(mm mm.time_variant = false) implies self.measurement_Transition → isEmpty())

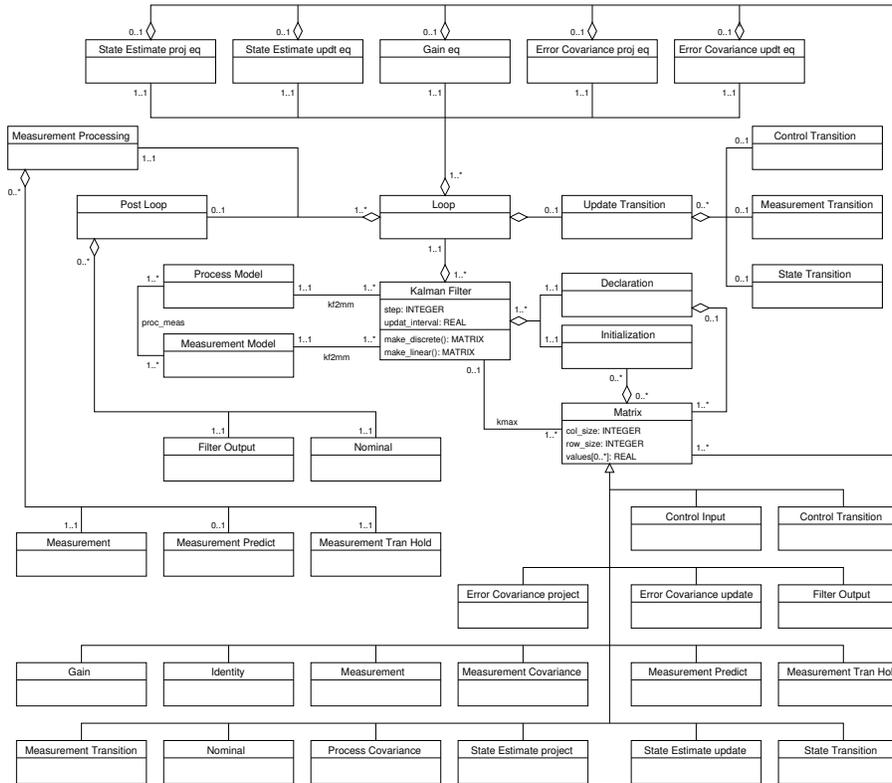


Fig. 2. AUTOFILTER's output CD domain model

The checking of AUTOFILTER's input equation specification and output filter intermediate code is made up of the following tasks: (1) checking the input specification against the input model (syntactic checking); (2) checking the output filter against the output model (syntactic checking); (3) mutual checking of the input and output semantic constraints (semantic checking).

Syntactic checking Syntactic checking of AUTOFILTER input equation specification and output filter intermediate code are conducted independently of each other. Checking of the input specification may be done before invoking AUTOFILTER, so as to ensure that any error in the output filter code is not as a result of an incorrect input specification. The syntactic checking includes:

1. Ensuring mandatory classes and associations are included in the input specification and output filter.
2. Ensuring that the multiplicities of the domain CDs are not violated.

3. Ensuring that all entities of the input specification and output filter are defined in the domain CDs.

Semantic checking Semantic checking, like the syntactic checking is conducted partly as a manually and partly as an automatic process and is intended to check aspects of the input specification and output filter that are other than structural. The semantic checking involves satisfying the constraints associated with the *meaning* of the input specification and output filter components.

Checking process The input specification and output filter must first be translated from its concrete syntax to the syntax of the DSML. This involves “lifting” from the concrete syntax to the domain-specific concepts defined by the DSML (Figures 1 and 2). This results in the generation of UML object diagrams of the input specification and output filter intermediate code.

The next step is the translation of the UML object diagrams to its *USE* format. The *USE* object diagram representation is then inputted to *USE* and both syntactic and semantic checking is automatically conducted. Any error reported from *USE* has to be checked against the object models to determine whether the manual translation processes introduced the error, or the error was as a result of modifications done to the program synthesizer.

5 Related Works

Our approach to input/output constraint-checking is a form of lightweight verification that emphasizes the role of modeling to highlight what is to be checked. The approach is a form of *product-oriented* verification in that we verify the result of the synthesizer rather than attempting to verify the synthesizer itself. There has been some other work on product-oriented certification. [10] checks the result of the program synthesizer AUTOBAYES³ for the violation of simple safety properties, such as safe array bounds access and absence of division by zero. The approach is to encode the safety properties as a set of rules (a safety policy) that can be used to generate verification conditions to be proved by a theorem prover. A similar approach is pursued in [7] in which term rewriting is used to check functional properties of the AUTOFILTER system. The product-oriented approach is derived from the ideas of proof-carrying code [6] in which a compiler is augmented with certificates of partial correctness of the object code generated. A related approach is that of run-time *result-checking* [8] in which correctness of a particular run of a system is checked at run-time rather than checking the correctness of the software itself — e.g., for a sorting algorithm, it is easier to check that a given sorted list is indeed sorted rather than check the correctness of the algorithm.

³ AUTOBAYES and AUTOFILTER are built on the same infrastructure but AUTOBAYES works in the domain of data analysis.

6 Conclusion

In this report a DSML-based technique has been presented for partial checking of the input specification and output code of program synthesizers. The rationale for this work lies in the need to be able to use automatic program generation with an acceptable degree of confidence. The technique presented in this report integrates formal constructs (e.g. OCL) with informal graphical modeling notation (e.g. UML) to create domain-specific modeling languages.

The technique was applied to a program synthesizer for state estimation problems from an industrial domain. The experience gained in this exercise demonstrates that this constraint-checking technique can provide a high level of confidence in the use of program synthesizers. But the processes used in the technique have to be nearly fully automated in order to derive the full benefits from the use of such techniques.

References

1. R. Akers, E. Kant, C. Randall, S. Steinberg, and R. Young. Scinapse: A problem-solving environment for partial differential equations. *IEEE Comp. Sci. and Eng.*, 4:32–42, 1997.
2. Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The genvoca model of software-system generators. *IEEE Software*, 11(5):89–94, September/October 1994.
3. Emanuel S. Grant. *Defining Domain-Specific Object-Oriented Modeling Languages as UML Profiles*. PhD thesis, Colorado State University, Ft. Collins, Colorado, USA, December 2002.
4. Object Management Group. *OMG Unified Modeling Language (UML) Specification*. Object management Group, Needham, Massachusetts, USA, uml 1.5 edition, March 2003.
5. R. Grover Brown and P.Y.C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering*. John Wiley & Sons, 1997.
6. G.C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages (POPL)*, 1997.
7. Grigore Rosu and Jonathan Whittle. Towards certifying domain-specific properties of synthesized code. In *Verification and Computational Logic (VCL'02)*, Pittsburgh, PA, USA, October 2002.
8. H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44:826–849, 1997.
9. David M. Weiss. Defining families: The Commonality Analysis. Technical Report Submitted to IEEE TSE, Lucent Technologies, 1998.
10. Michael Whalen, Johann Schumann, and Bernd Fischer. Synthesizing certified code. In *Formal Methods Europe*, pages 431–450. Springer, 2002.
11. J. Whittle, J. van Baalen, J. Schumann, P. Robinson, , T. Pressburger, J Penix, P. Oh, M. Lowry, and G. Brat. Amphion/NAV: Deductive synthesis of state estimation software. In *Proceedings of Conference on Automated Software Engineering (ASE01)*, San Diego, CA, USA, 2001.