Proceedings of the ASME 2011 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference
IDETC/CIE 2011
August 28-31, 2011, Washington, DC, USA

DETC2011-48530

A PROGRAMMING LANGUAGE APPROACH TO PARAMETRIC CAD DATA EXCHANGE

John Altidor
Jack Wileden
Department of Computer Science
University of Massachusetts
Amherst, MA 01003
Email:
{jaltidor,jack}@cs.umass.edu

Jeffrey McPherson
Ian Grosse
Sundar Krishnamurty
Department of Mechanical Engineering
University of Massachusetts
Amherst, MA 01003
Email: jdmcpher@student.umass.edu
{grosse,skrishna}@ecs.umass.edu

Felicia Cordeiro
Audrey Lee-St. John
Department of Computer Science
Mount Holyoke College
South Hadley, MA 01075
Email:
{corde20f,astjohn}@mtholyoke.edu

ABSTRACT

Data exchange between different computer-aided design (CAD) systems is a major problem inhibiting information integration in collaborative engineering environments. Existing CAD data format standards such as STEP and IGES enable geometric data exchange. However, they ignore construction history, features, constraints, and other parametric-based CAD data. As a result, they are inadequate for supporting modification, extension and other important higher-level functionality when accessing an imported CAD model from another CAD system. Achieving such higher-level functionality therefore often requires a time-consuming, error-prone, tedious process of manually recreating the model in the target CAD system.

Based on techniques adapted from programming language research, this paper presents an approach to exchanging parametric data between CAD systems using formally-defined *conversion semantics*. We have demonstrated the utility of our approach by developing a prototype implementation that automates the conversion of 2D sketches between two popular CAD systems: Pro/ENGINEER and SolidWorks. We present examples showing that our approach is able to accurately convert parametric CAD data even in cases where models were constructed using operations from the source CAD system that have no direct counterpart in the target CAD system. Although the case study focuses on 2D interoperability, our approach provides *formal foundations* for supporting 3D and semantic interoperability between CAD systems.

1 INTRODUCTION

Modern trends toward increasing globalization and product complexity have increased the need for collaboration and communication of product design. One of the major cost elements associated with product design is the precious time engineers spend on data translation between different computer-aided design (CAD) formats, data integration between different systems, routine data reprocessing in different application scenarios, redesign due to loss of information, and error correction due to human errors in the above processes. Existing standard CAD data formats such as IGES and STEP focus on geometric data, but they do not preserve parametric-based CAD data such as construction history, features, and constraints. Hence, design-intent, semantic-level information, and the ability to perform high-level modification is lost when exchanging models between CAD systems. Applying ontologies has become a popular approach to preserving design intent. Because of the popularity of the ontological approach, Section 6.1 compares our approach to ontological approaches in detail.

This paper presents our approach to parametric CAD data exchange applying techniques from *programming language theory*. We model CAD systems as programming languages. CAD models correspond to programs in the languages modeling the CAD systems. This paper makes several practical and theoretical contributions.

Practical Contributions: We implemented software that not only converts between intermediate representations of CAD models but also converts between proprietary CAD formats, namely Pro/ENGINEER (Pro/E) and SolidWorks (SW). We present algorithms that can convert Pro/E 2D sections containing elements with no direct counterpart in SW. Moreover, our approach supports not only 1-1 mappings but also many-to-many mappings leading to more accurate translations. Our algorithms are implemented in software automating the conversions.

We provide open XML formats that (1) humans can read and edit and (2) are linked to the semantics of Pro/E and SW by our software. One could create a CAD model in the proprietary Pro/E or SW format *without using the graphical user interfaces* of CAD systems by applying our software to an XML representation of the CAD model. Other CAD interoperability researchers can apply and automate their logic over our open, easy-to-parse XML formats to experiment with their approach and not worry about learning CAD APIs. We present example conversions in Section 5 from applying our approach to CAD models demonstrating that our approach can convert CAD models accurately and preserve design intent.

Theoretical Contributions: We present a rigorous model of CAD systems that enables formal semantics of parametric CAD data and precise analyses. We formally define our algorithm for converting models between CAD systems. Our algorithms are defined by mathematical specifications, not by code or other informal descriptions. Our formalization is not dependent on technologies such as the semantic web.

Section 2 provides a brief introduction to programming language theory. Section 3 presents our approach to parametric CAD data exchange. Section 4 discusses the implementation of our approach. Section 5 presents our empirical results. Section 6 compares our approach to popular related approaches. Section 7 summarizes.

2 INTRODUCTION TO PROGRAMMING LANGUAGE FOUNDATIONS

Parametric-based representations of CAD models are extensions of constructive solid geometry describing CAD models as a combination of primitives that are combined using operations. Each operation is defined by describing its operands and how it relates with its operands. Most related work fits into this description. According to [1]: "E-Rep [2,3] distinguishes between generated features, datum features, and modifying features; it regards a CAD model as being built entirely by a sequence of feature insertion, modification, and deletion operations. The ENGEN Data Model (EDM) [4,5] extended STEP's current explicit entity representation by adding some predefined local features such as round and chamfer in a bottom-up approach."

Consider creating the model shown in Figure 1. A possible specification of the model is that it is a cylinder with radius 2cm and height 4cm, a cube with side-length 3cm, and the centers of the cylinder and cube are separated by exactly 6cm.

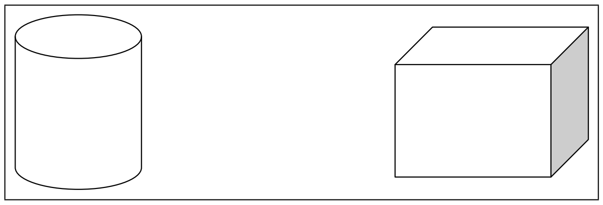

FIGURE 1. Simple CAD model example

This model can be created by the following process:

1. Create a 2-dimensional (2D) circle with radius of 2cm.
2. Extrude the circle with depth of 4cm.
3. Create a cube with side-length 3cm.
4. Add the constraint to make centers of the cylinder and cube be separated by exactly 6cm.

2.1 Syntax

Using the terminology and notation of programming language foundations, the model in Figure 1 can be represented by the following *abstract syntax tree* (AST):

```
Constraint(Extrude(Circle(2), 4), Cube(3), 6)
```

ASTs are also more simply called *terms*. Let T denote the above term. ASTs are mathematical expressions that represent a composition of *operators* or *nodes*. Each AST has the form:

$$operator(operand_1, operand_2, \ldots, operand_n),$$

where each operand is also an AST and the operator is a root node of these ASTs. AST nodes are basically operators that take a specified number of operands. An operator can take zero operands; in this case, the parentheses after the operator are typically not written. For instance, the node `6` in the AST T above could have been written as `6()`.

The "tree" in abstract syntax tree comes from the fact that ASTs can be drawn as a tree diagram; Figure 2 shows the tree diagram of T. Operators are nodes in the tree. Operands of an operator are subtrees extending from the operator node in the tree.

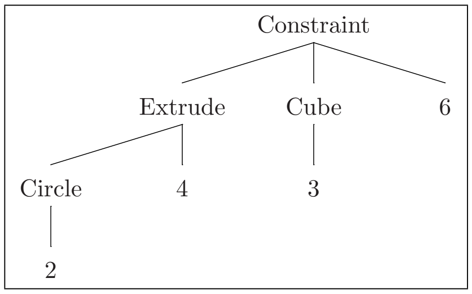

FIGURE 2. Tree diagram of Constraint(Extrude(Circle(2), 4), Cube(3), 6)

ASTs are used extensively in programming languages. The *abstract syntax* of a language determines the set of nodes available in the language. A language's abstract syntax is defined by its *formal grammar*, which specifies *grammar/productions rules* specifying how ASTs are constructed. A *non-terminal symbol N* denotes a set of ASTs specified by a grammar rule, where N is on the left-hand side of the $::=$. Each grammar rule has the form "$A ::= B_1 \mid B_2 \mid \dots \mid B_n$". Each B_i is either a non-terminal symbol or a specific AST. A is the set of ASTs that is the union of ASTs in $B_1, B_2, \dots, B_n$. Figure 3 shows an example grammar for a simple language of arithmetic expressions; we call this language *MiniLang*.

Category	**Item**	**AST Node**
Expression	e	$::=$ `num[`n`]` $\mid x$
		$\mid$ `+(`e_1`, `e_2`)`
		$\mid$ `let(`x`, `e_1`, `e_2`)`

FIGURE 3. Grammar of *MiniLang* language. n denotes a sequence of digits. x denotes a sequence of letters.

The only non-terminal symbol in *MiniLang* is e, which denotes the set of ASTs that are expressions. Expressions can be the following:

1. Numbers: `num[`n`]`, where n is a sequence of digits.
2. Variable names: x.
3. The sum of two subexpressions: `+(`e_1`, `e_2`)`.
4. A let expression where the subexpression e_2 is evaluated in a context that has variable x bound to the value of e_1.

Three example ASTs specified by this grammar are below:

1. `+(num[3], num[4])`
2. `+(+(num[6], num[2]), num[1])`
3. `let(daysPerWeek, num[7], +(num[1], daysPerWeek))`

This model of a language allows one to formally define its semantics and analyze properties. An important category of language semantics is *dynamic semantics* also called operational semantics. Dynamic semantics inductively define how to *evaluate* ASTs; specifically, it defines a *transition system* between ASTs, where the *values* that ASTs evaluate to are the *final states* of the system. Section 3 shows how to apply dynamic semantics to CAD data exchange. First, we start with a simpler example defining dynamic semantics over *MiniLang*.

2.2 Dynamic Semantics

Dynamic semantics are defined by a set of *inference rules* that define a transition system for evaluating *MiniLang* expressions. Inferences rules have the following form:

$$\frac{\overbrace{J_1 \quad J_2 \quad \dots \quad J_n}^{\textit{premises}}}{\underbrace{J}_{\textit{conclusion}}} \;\texttt{Rule Label}$$

Each J_i is a proposition or *judgment*. If all of the judgments of the premise (J_i's) are true, then the conclusion judgment J is true. Rules with no premises are axioms because the conclusion is true under any conditions. The dynamic semantics of *MiniLang* define a transition system for evaluating *MiniLang* expressions. In order to know when we are done evaluating an expression to a *value*, we need to define what values are. Values in *MiniLang* are single numbers.

$$\frac{}{\texttt{num[}n\texttt{] value}}$$

The inductive definition of the transition relation for evaluating expressions is shown in Figure 4. Rules `D.1-3` define how to evaluate additions. Rule `D.3` says that the addition of two single numbers *steps to* (transitions to) a number that is the sum of those two numbers. Rule `D.1` says if an evaluation step can be performed on the left subexpression, then the addition expression steps to an expression that is the same except with the step performed on the left subexpression. Once we are done evaluating the left subexpression to a number, rule `D.2` allows us to evaluate the right subexpression. The notation $e_2[x/e_1]$ denotes the expression obtained from e_2 by substituting e_1 for every occurrence of x in e_2. Rule `D.4` says we evaluate the expression (e_1) that will be bound to the variable of the `let` expression. Once e_1 becomes a value, then rule `D.5` says we can evaluate the body (e_2) of the `let` expression by replacing the variable with that value. An example expression evaluation is shown Figure 5.

$$\frac{e_1 \mapsto e_1'}{\texttt{+(}e_1\texttt{, } e_2\texttt{)} \mapsto \texttt{+(}e_1'\texttt{, } e_2\texttt{)}}\ \texttt{D.1}$$

$$\frac{e_2 \mapsto e_2'}{\texttt{+(num[}n_1\texttt{], } e_2\texttt{)} \mapsto \texttt{+(num[}n_1\texttt{], } e_2'\texttt{)}}\ \texttt{D.2}$$

$$\frac{}{\texttt{+(num[}n_1\texttt{], num[}n_2\texttt{])} \mapsto \texttt{num[}n_1+n_2\texttt{]}}\ \texttt{D.3}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{let(}x\texttt{, } e_1\texttt{, } e_2\texttt{)} \mapsto \texttt{let(}x\texttt{, } e_1'\texttt{, } e_2\texttt{)}}\ \texttt{D.4}$$

$$\frac{e_1\ \texttt{value}}{\texttt{let(}x\texttt{, } e_1\texttt{, } e_2\texttt{)} \mapsto e_2[x/e_1]}\ \texttt{D.5}$$

FIGURE 4. Dynamic Semantics of *MiniLang*

`let(x, +(2,1), +(x,5))`	
$\mapsto$ `let(`x`, 3, +(x,5))`	by rules `D.4` and `D.3`
	because `+(2,1)` $\mapsto$ `3`
$\mapsto$ `+(x,5)`$[x/3]$	by rule `D.5`
	because `3 value`
$=$ `+(3,5)`	applying substitution operation
$\mapsto 8$	by rule `D.3`

FIGURE 5. Evaluation of expression: `let(x, +(2,1), +(x,5))`. The `num[ ]` tags on numbers are omitted for brevity.

3 LANGUAGE APPROACH TO CAD DATA EXCHANGE

This section presents our programming language approach to parametric CAD data exchange. We focus on converting 2D sections from Pro/E to 2D sections in SW. We do so by converting between two programming languages, one modeling a subset of Pro/E and the other modeling a subset of SW. Section 3.1 presents *ProLang*, the language modeling Pro/E. Section 3.2 presents *SWLang*, the language modeling SW. Section 3.3 compares the two language models. Section 3.4 describes our conversion from Pro/E to SW via *conversion semantics*. Section 3.5 discusses how to extend this general approach to convert CAD elements not mentioned in this paper.

3.1 Pro/ENGINEER Programming Language: *ProLang*

Figure 6 defines the abstract syntax for a programming language modeling a subset of functionality provided by the Pro/E CAD system for creating two-dimensional sections. A program in this language corresponds to a 2D section in Pro/E. The symbol $\Rightarrow$ designates that S^P is the start symbol of the language. The start symbol of a language specifies that in order for an AST to be in the grammar of the language, the AST must be in the set of ASTs denoted by the start symbol. Hence, the root operator of any AST specified in the grammar of *ProLang* must be `Pro2D`. This signifies that an AST in *ProLang* must represent an entire 2D section. For instance, the AST `Point`^{P}`(4.3, 5.2, 1)` by itself is not in the grammar of *ProLang*; it only represents a single point, not an entire 2D section.

Category	**Item**	**Production**
$\Rightarrow$ *Section*^{P}	S^P	$::=$ `Pro2D(`$\overline{E^P}$`, `$\overline{C^P}$`, `$\overline{D^P}$`)`
Entity^P	E^P	$::= L^P \mid Q^P$
Line^P	L^P	$::=$ `Line`^{P}`(`id^P`, `b`, `P_1^P`, `P_2^P`)`
EntityPoint	Q^P	$::=$ `EntPoint(`id^P`, `P^P`)`
SimplePoint^P	P^P	$::=$ `Point`^{P}`(`r_x`, `r_y`, `id^P`)`
Constraint^P	C^P	$::=$ `SamePoint`^{P}`(`id^P`, `P_1^P`, `P_2^P`)`
		$\mid$ `HorizontalConstraint`^{P}`(`id^P`, `L^P`)`
		$\mid$ `VerticalConstraint`^{P}`(`id^P`, `L^P`)`
		$\mid$ `PntOnEnt`^{P}`(`id^P`, `L^P`, `P^P`)`
Dimension^P	D^P	$::=$ `LineDim`^{P}`(`id^P`, `r`, `L^P`)`
		$\mid$ `LinePointDim`^{P}`(`id^P`, `r`, `L^P`, `P^P`)`
Identifier^P	id^P	$::= n$
Boolean	b	$::=$ `true` $\mid$ `false`
Integer	n	$::= \ldots$
Real	r	$::= \ldots$

FIGURE 6. Syntax of Pro/E 2D Sections

Notice that many operators are annotated with a P superscript. This annotation is used to clarify that it is a *ProLang* operator. Later in this paper, there will be mathematical expressions involving symbols from *ProLang* and from *SWLang*, the programming language modeling SW. The annotations will clarify which language the symbols are from.

The notation $\overline{A}$ denotes a possibly empty list, $[A_1, A_2, \ldots, A_n]$. A Pro/E 2D section S^P is a term `Pro2D(`$\overline{E^P}$`, `$\overline{C^P}$`, `$\overline{D^P}$`)` composed of a list of entities $\overline{E^P}$, a list of constraints $\overline{C^P}$, and a list of dimensions $\overline{D^P}$. An entity E^P is a line L^P or an entity point Q^P. An entity point is a term `EntPoint(`id^P`, `P^P`)` composed of an identifier id^P and a simple point P^P. A simple point is a term `Point`^{P}`(`r_x`, `r_y`, `id^P`)` composed of two real numbers r_x and r_y defining the x- and y-coordinates, respectively, and an identifier id^P that does not identify the simple point but the enclosing entity. All simple points are parameters of some enclosing entity. A line is a term `Line`^{P}`(`id^P`, `b`, `P_1^P`, `P_2^P`)` composed of an identifier id^P, a boolean value b signaling whether or not it is an axis, the first end point of the line P_1^P, and the second end point of the line P_2^P.

Four different types of constraints are available. Every constraint has an identifier. The $\texttt{SamePoint}^P(id^P,\ P_1^P,\ P_2^P)$ term specifies that points P_1^P and P_2^P should be at the same location. The $\texttt{HorizontalConstraint}^P(id^P,\ L^P)$ term specifies that the end points of line L^P should have the same y-coordinate. The $\texttt{VerticalConstraint}^P(id^P,\ L^P)$ term specifies that the end points of line L^P should have the same x-coordinate. The $\texttt{PntOnEnt}^P(id^P,\ L^P,\ P^P)$ term specifies that point P^P is required to be somewhere on line L^P.

The $\texttt{LineDim}^P(id^P,\ r,\ L^P)$ term specifies a line dimension such that the end points of L^P should be separated by distance r. The $\texttt{LinePointDim}^P(id^P,\ r,\ L^P,\ P^P)$ term specifies that the shortest distance between line L^P and point P^P should be r. Lastly, an identifier id^P is an integer n.

3.2 SolidWorks Programming Language: *SWLang*

Figure 7 shows a programming language modeling a subset of functionality provided by the SW CAD system for creating two-dimensional sections. This subset is similar to the subset of Pro/E functionality modeled by *ProLang*. The start symbol is S^S, a non-terminal symbol representing a 2D section in SW. A SW 2D section S^S is a term $\texttt{SW2D}(\overline{E^S},\ \overline{C^S},\ \overline{D^S})$ composed of entities $\overline{E^S}$, constraints $\overline{C^S}$, and dimensions $\overline{D^S}$.

Category	**Item**	**Production**
$\Rightarrow$ *Section*^{S}	S^S	$::=\ \texttt{SW2D}(\overline{E^S},\ \overline{C^S},\ \overline{D^S})$
Entity^S	E^S	$::=\ L^S \mid P^S$
Line^S	L^S	$::=\ \texttt{Line}^S(ide,\ P_1^S,\ P_2^S)$
Point^S	P^S	$::=\ \texttt{Point}^S(ide,\ r_x,\ r_y,\ r_z)$
Constraint^S	C^S	$::=\ \texttt{Coincident}^S(P_1^S,\ P_2^S)$
		$\mid\ \texttt{HorizontalConstraint}^S(L^S)$
		$\mid\ \texttt{VerticalConstraint}^S(L^S)$
Dimension^S	D^S	$::=\ \texttt{LineDim}^S(idd,\ r,\ E_1^S,\ E_2^S)$
		$\mid\ \texttt{HorLineDim}^S(idd,\ r,\ E_1^S,\ E_2^S)$
		$\mid\ \texttt{VerLineDim}^S(idd,\ r,\ E_1^S,\ E_2^S)$
EntityId^S	ide	$::=\ \texttt{ide}(n_1,\ n_2)$
DimensionId^S	idd	$::=\ n$
Integer	n	$::=\ \ldots$
Real	r	$::=\ \ldots$

FIGURE 7. Syntax of SW 2D Sections

A SW entity E^S is a line L^S or a point P^S. Each entity has an entity identifier, *ide*, that is a pair of integers. A line is a $\texttt{Line}^S(ide,\ P_1^S,\ P_2^S)$ term specifying two end points P_1^S and P_2^S. A point is a $\texttt{Point}^S(ide,\ r_x,\ r_z,\ r_z)$ term consisting of 3 real numbers r_x, r_y, and r_z specifying the x-, y-, and z-coordinates, respectively; SW allows sketches to be specified in three dimensional space.

Three constraints are available. The $\texttt{Coincident}^S(P_1^S,\ P_2^S)$ term specifies that points P_1^P and P_2^P should be at the same location. The $\texttt{HorizontalConstraint}^S(L^S)$ term specifies that the end points of line L^S should have the same y-coordinate; $\texttt{VerticalConstraint}^S(L^S)$ specifies that the end points of L^S should have the same x-coordinate.

The $\texttt{LineDim}^S(idd,\ r,\ E_1^S,\ E_2^S)$ term specifies that the shortest distance between entities E_1^S and E_2^S should be r. The $\texttt{HorLineDim}^S$ and $\texttt{VerLineDim}^S$ terms specify that the shortest horizontal and vertical distance, respectively, between entities E_1^S and E_2^S should be r. Lastly, a dimension identifier *idd* is an integer n.

3.3 *ProLang* and *SWLang* Comparison

Languages *ProLang* and *SWLang* are very similar in structure; they model roughly similar subsets of the 2D sketch functionality of Pro/E and SW. For example, the only types of entities for both languages are lines and points. There are many other similarities but also significant differences that must be handled to convert models between the two languages. First, points in Pro/E 2D sections are defined with two coordinates, but points in SW 2D sections are defined with three coordinates. Pro/E has two levels of points. As we'll see in Section 4.1, $\texttt{EntPoint}$'s enable constraints and dimensions to refer to a $\texttt{Point}^P$ that is not part of a more geometrically complex structure (e.g., $\texttt{Line}^P$). Pro/E lines can be axes, but SW lines cannot and do not take a boolean parameter. Line dimensions in Pro/E are defined using lines, but line dimensions in SW are defined with two entities. SW also allows horizontal and vertical line dimensions that specify only the horizontal and vertical separation. The most significant difference is that the *ProLang* constraint $\texttt{PntOnEnt}^P$ has no direct equivalent in *SWLang*. Although it might seem that any *ProLang* section containing a $\texttt{PntOnEnt}^P$ constraint could not be converted into an equivalent section in SW, we present an example *ProLang* section containing $\texttt{PntOnEnt}^P$ constraints that has an equivalent *SWLang* representation. Our conversion approach is able to convert such Pro/E sections. Section 3.4 describes our approach to converting from *ProLang* to *SWLang*.

3.4 Conversion from Pro/ENGINEER to SolidWorks

In this section, we describe our algorithm for converting Pro/E 2D sections to SW 2D sections using *conversion semantics*. First, we need to define notation and operations over lists.

3.4.1 List Operations

The notation $[A_1, A_2, \ldots, A_n]$ denotes a list, where the i^{th} element of the list is A_i, and $[]$ represents an empty list. The concatenation operator $+$ is used to concatenate two list; formally:

$$[A_1,A_2,\ldots,A_n]+[B_1,B_2,\ldots,B_m]=[A_1,A_2,\ldots,A_n,B_1,B_2,\ldots,B_m] \tag{1}$$

The subtraction operator $-$ is used to remove a sublist of elements from a list; formally:

$$[A_1,A_2,\ldots,A_n]-[\,]=[A_1,A_2,\ldots,A_n] \tag{2}$$

$$[\,]-B=[\,] \tag{3}$$

$$\begin{aligned}&[A_1,A_2,\ldots,A_n]-[B_1,B_2,\ldots,B_m]\\ &=\begin{cases}[A_2,\ldots,A_n]-[B_1,\ldots,B_{i-1},B_{i+1},\ldots,B_m], \text{ if } A_1=B_i\\ [A_1]+([A_2,\ldots,A_n]-[B_1,\ldots,B_m]), \text{ otherwise}\end{cases}\end{aligned} \tag{4}$$

For example, $[3,5,4,2]-[1,2,3]=[5,4]$. If A is a list, the notation $a\in A$ denotes a is an element in A. The notation $A\subseteq B$ denotes A is a sublist of B; formally:

$$[\,]\subseteq B=\textbf{true} \tag{5}$$

$$[A_1,A_2,\ldots,A_n]\subseteq B=\begin{cases}[A_2,\ldots,A_n]\subseteq B-[A_1], \text{ if } A_1\in B\\ \textbf{false}, \text{ otherwise}\end{cases} \tag{6}$$

For example, $[2,1]\subseteq[1,2,3]$ but $[2,1]\not\subseteq[1,3]$. Now we define the $+$, $-$, and $\subseteq$ operations for 2D sections.

$$\begin{aligned}&\texttt{Pro2D(}\overline{E_1^P}\texttt{, }\overline{C_1^P}\texttt{, }\overline{D_1^P}\texttt{)}+\texttt{Pro2D(}\overline{E_2^P}\texttt{, }\overline{C_2^P}\texttt{, }\overline{D_2^P}\texttt{)}\\ &=\texttt{Pro2D(}\overline{E_1^P}+\overline{E_2^P}\texttt{, }\overline{C_1^P}+\overline{C_2^P}\texttt{, }\overline{D_1^P}+\overline{D_2^P}\texttt{)}\end{aligned} \tag{7}$$

$$\begin{aligned}&\texttt{Pro2D(}\overline{E_1^P}\texttt{, }\overline{C_1^P}\texttt{, }\overline{D_1^P}\texttt{)}-\texttt{Pro2D(}\overline{E_2^P}\texttt{, }\overline{C_2^P}\texttt{, }\overline{D_2^P}\texttt{)}\\ &=\texttt{Pro2D(}\overline{E_1^P}-\overline{E_2^P}\texttt{, }\overline{C_1^P}-\overline{C_2^P}\texttt{, }\overline{D_1^P}-\overline{D_2^P}\texttt{)}\end{aligned} \tag{8}$$

$$\frac{\overline{E_1^P}\subseteq\overline{E_2^P}\quad\overline{C_1^P}\subseteq\overline{C_2^P}\quad\overline{D_1^P}\subseteq\overline{D_2^P}}{\texttt{Pro2D(}\overline{E_1^P}\texttt{, }\overline{C_1^P}\texttt{, }\overline{D_1^P}\texttt{)}\subseteq\texttt{Pro2D(}\overline{E_2^P}\texttt{, }\overline{C_2^P}\texttt{, }\overline{D_2^P}\texttt{)}}\ \texttt{SUBSET}$$

The operations $+$, $-$, and $\subseteq$ are defined on SW 2D sections analogously.

3.4.2 Conversion Semantics

We describe our algorithm for converting from Pro/E 2D sections to SW 2D sections using conversion semantics (a type of dynamic semantics). Conversion semantics define a *transition* system on a set of *states*. A state in our transition system is a pair (S^P, S^S), where S^P is a Pro/E 2D section and S^S is a SW 2D section. S^P represents the remaining Pro/E section to convert. S^S represents what the conversion has produced so far. Transition systems have designated *starting states* to specify which states they start in. Transition systems complete when they reach one of their designated *final states*. Starting states in our transition system have the form:

$$\left(\texttt{Pro2D(}\overline{E^P}\texttt{, }\overline{C^P}\texttt{, }\overline{D^P}\texttt{), SW2D([], [], [])}\right) \tag{9}$$

This signals that we start with an arbitrary Pro/E 2D section and an empty SW 2D section. Final states in our transition system have the form:

$$\left(\texttt{Pro2D([], [], []), SW2D(}\overline{E^S}\texttt{, }\overline{C^S}\texttt{, }\overline{D^S}\texttt{)}\right) \tag{10}$$

This signals that we end when there are no more entities, constraints, or dimensions in the Pro/E 2D section to convert.

We will define an equivalence relation $S^P\equiv S^S$ between simple Pro/E and SW 2D sections. The equivalences between the simple sections will allow us to convert between more complex sections. The key transition rule of our conversion semantics is below.

$$\frac{S_2^P\subseteq S_1^P\quad S_2^P\equiv S_2^S}{(S_1^P,\,S_1^S)\mapsto(S_1^P-S_2^P,\,S_1^S+S_2^S)}\ \texttt{TRANS}$$

The `TRANS` rule says that if we have identified a subsection S_2^P of the Pro/E 2D section S_1^P that we know is equivalent to the SW 2D section S_2^S, then a next step in our conversion can be removing subsection S_2^P from our remaining work and outputting the equivalent SW subsection S_2^S. The premise $S_2^P\subseteq S_1^P$ is needed so that we do not introduce "new stuff" to the output SW section that has no counterpart in the original section from Pro/E.

3.4.3 Pro/E and SW Equivalences

This section presents equivalence relations that can be used to convert Pro/E 2D sections to SW 2D sections. In addition to converting using one-to-one mappings, our approach allows converting between multiple combinations of entities, constraints, and dimensions. First, we present our one-to-one mapping approach.

We define the partial function $conv_E : E^P \dashrightarrow E^S$, which maps Pro/E entities to equivalent SW entities. The partial functions $conv_C : C^P \dashrightarrow C^S$ and $conv_D : D^P \dashrightarrow D^S$ are defined similarly over constraints and dimensions, respectively. The function $conv_O$ converts identifiers and simple points. We assume an implicit context that is available to the conversion functions, *section**, which just denotes the enclosing 2D section of the entity, constraint, or dimension of interest. *section** is used to ensure that no two distinct SW elements created by the the conversion contain the same identifier. *section** can also be used as a function mapping Pro/E identifiers to their corresponding *ProLang* terms. Now we define the conversion functions over elements that can be mapped one-to-one.

Converting Entities:

$$\begin{aligned} &conv_E(\mathtt{Line}^P\mathtt{(}id^P\mathtt{,}\ \mathbf{false}\mathtt{,}\ P_1^P\mathtt{,}\ P_2^P\mathtt{))} \\ &= \mathtt{Line}^S\mathtt{(}conv_O(id^P)\mathtt{,}\ conv_O(P_1^P)\mathtt{,}\ conv_O(P_2^P)\mathtt{)} \end{aligned} \tag{11}$$

$$conv_E(\mathtt{EntPoint(}id^P\mathtt{,}\ P^P\mathtt{))} = conv_O(P^P) \tag{12}$$

Converting Identifiers and Simple Points:

$$conv_O(n) = \mathtt{ide(}n\mathtt{,\ 0)} \tag{13}$$

$$\begin{aligned} &conv_O(\overbrace{\mathtt{Point}^P\mathtt{(}r_x\mathtt{,}\ r_y\mathtt{,}\ id^P\mathtt{)}}^{\boxed{P^P}}) \\ &= \mathtt{Point}^S\mathtt{(ide(}id^P\mathtt{,}\ n\mathtt{),}\ r_x\mathtt{,}\ r_y\mathtt{,\ 0),}\ \text{where} \end{aligned} \tag{14}$$

$$n = \begin{cases} 0 & \text{if } section^*(id^P) = \mathtt{EntPoint(}id^P\mathtt{,}\ \boxed{P^P}\mathtt{)} \\ 1 & \text{if } section^*(id^P) = \mathtt{Line}^P\mathtt{(}id^P\mathtt{,}\ b\mathtt{,}\ \boxed{P^P}\mathtt{,}\ P_2^P\mathtt{)} \\ 2 & \text{if } section^*(id^P) = \mathtt{Line}^P\mathtt{(}id^P\mathtt{,}\ b\mathtt{,}\ P_1^P\mathtt{,}\ \boxed{P^P}\mathtt{)} \end{cases}$$

The second number n in the pair of numbers of the SW entity identifier $\mathtt{ide(}id^P\mathtt{,}\ n\mathtt{)}$ is determined by the enclosing entity of the simple point P^P and how P^P was being passed as a parameter of the enclosing entity. If P^P is the parameter of an `EntPoint`, then $n = 0$. If P^P is the first end point of a line, then $n = 1$. If P^P is the second end point of a line but not the first, then $n = 2$.

Converting Constraints:

$$\begin{aligned} &conv_C(\mathtt{SamePoint}^P\mathtt{(}id^P\mathtt{,}\ P_1^P\mathtt{,}\ P_2^P\mathtt{))} \\ &= \mathtt{Coincident}^S\mathtt{(}conv_O(P_1^P)\mathtt{,}\ conv_O(P_2^P)\mathtt{)} \end{aligned} \tag{15}$$

$$\begin{aligned} &conv_C(\mathtt{HorizontalConstraint}^P\mathtt{(}id^P\mathtt{,}\ L^P\mathtt{))} \\ &= \mathtt{HorizontalConstraint}^S\mathtt{(}conv_E(L^P)\mathtt{)} \end{aligned} \tag{16}$$

$$\begin{aligned} &conv_C(\mathtt{VerticalConstraint}^P\mathtt{(}id^P\mathtt{,}\ L^P\mathtt{))} \\ &= \mathtt{VerticalConstraint}^S\mathtt{(}conv_E(L^P)\mathtt{)} \end{aligned} \tag{17}$$

Converting Dimensions:

$$\begin{aligned} &conv_D(\mathtt{LineDim}^P\mathtt{(}id^P\mathtt{,}\ r\mathtt{,}\ \mathtt{Line}^P\mathtt{(}id^P\mathtt{,}\ b\mathtt{,}\ P_1^P\mathtt{,}\ P_2^P\mathtt{)))} \\ &= \mathtt{LineDim}^S\mathtt{(}conv_O(id^P)\mathtt{,}\ r\mathtt{,}\ conv_O(P_1^P)\mathtt{,}\ conv_O(P_2^P)\mathtt{)} \end{aligned} \tag{18}$$

For converting $\mathtt{LinePointDim}^P\mathtt{(}id^P\mathtt{,}\ r\mathtt{,}\ L^P\mathtt{,}\ P^P\mathtt{)}$ terms, if L^P is an x-axis or y-axis, then we map the dimension to a $\mathtt{VerLineDim}^S$ or $\mathtt{HorLineDim}^S$ term as shown below, where **origin** denotes a SW point at the origin. Recall that the second parameter of a Pro/E line term is a boolean value signaling if it is an axis line.

$$\begin{aligned} &conv_D(\mathtt{LinePointDim}^P\mathtt{(}id^P\mathtt{,}\ r\mathtt{,}\ L^P\mathtt{,}\ P^P\mathtt{))} \\ &= \mathtt{VerLineDim}^S\mathtt{(}conv_O(id^P)\mathtt{,}\ r\mathtt{,}\ \mathbf{origin}\mathtt{,}\ conv_O(P^P)\mathtt{)}, \\ &\text{if } L^P \text{ is the } x\text{-axis} \end{aligned} \tag{19}$$

$$\begin{aligned} &conv_D(\mathtt{LinePointDim}^P\mathtt{(}id^P\mathtt{,}\ r\mathtt{,}\ L^P\mathtt{,}\ P^P\mathtt{))} \\ &= \mathtt{HorLineDim}^S\mathtt{(}conv_O(id^P)\mathtt{,}\ r\mathtt{,}\ \mathbf{origin}\mathtt{,}\ conv_O(P^P)\mathtt{)}, \\ &\text{if } L^P \text{ is the } y\text{-axis} \end{aligned} \tag{20}$$

For converting $\mathtt{LinePointDim}^P$ dimensions between non-axis lines and points, we use an auxilary function such that given a line L^P and a point P^P, $project(L^P, P^P) = P^S$, where P^S is the ($conv_O$ of the) point on line L^P that P^P projects to.

$$\begin{aligned} &conv_D(\mathtt{LinePointDim}^P\mathtt{(}id^P\mathtt{,}\ r\mathtt{,}\ L^P\mathtt{,}\ P^P\mathtt{))} \\ &= \mathtt{LineDim}^S\mathtt{(}conv_O(id^P)\mathtt{,}\ r\mathtt{,}\ project(L^P, P^P)\mathtt{,}\ conv_O(P^P)\mathtt{)} \end{aligned} \tag{21}$$

Lastly, we define the following equivalence rules to take advantage of the conversion functions.

$$\frac{conv_E(E_P) = E_S}{\mathtt{Pro2D([}E_P\mathtt{],\ [],\ [])} \equiv \mathtt{SW2D([}E_S\mathtt{],\ [],\ [])}}\ \mathtt{E.1}$$

$$\frac{conv_E(C_P) = C_S}{\mathtt{Pro2D([],\ [}C_P\mathtt{],\ [])} \equiv \mathtt{SW2D([],\ [}C_S\mathtt{],\ [])}}\ \mathtt{E.2}$$

$$\frac{conv_E(D_P) = D_S}{\mathtt{Pro2D([],\ [],\ [}D_P\mathtt{])} \equiv \mathtt{SW2D([],\ [],\ [}D_S\mathtt{])}}\ \mathtt{E.3}$$

An Example of a Many-to-Many Conversion: The $conv_C$ conversion does not map the $\mathtt{PntOnEnt}^P$ constraint to any *SWLang* constraint because there is no equivalent constraint in *SWLang* to map the constraint to. However, there are Pro/E 2D sections containing $\mathtt{PntOnEnt}^P$ constraints that can be mapped to equivalent SW 2D sections. The following equivalence rule recognizes one such situation. This situation occurs when two $\mathtt{PntOnEnt}^P$ constraints are used to constrain the intersection of two lines to a certain point. This behavior can be simulated in SW, with the help of an auxiliary function *intersect*.

$$\frac{intersect(L_1^P, L_2^P) = P_{int}^P}{\begin{array}{c} \mathtt{Pro2D([],\ [PntOnEnt}^P\mathtt{(}id_1^P\mathtt{,}\ L_1^P\mathtt{,}\ \boxed{P^P}\mathtt{),} \\ \mathtt{PntOnEnt}^P\mathtt{(}id_2^P\mathtt{,}\ L_2^P\mathtt{,}\ \boxed{P^P}\mathtt{)],\ [])} \\ \equiv \mathtt{SW2D([],} \\ \mathtt{[Coincident}^S\mathtt{(}conv_O(P_{int}^P)\mathtt{,}\ conv_O(\boxed{P^P})\mathtt{)],\ [])} \end{array}}\ \mathtt{INTSECT}$$

The *intersect* function is a partial function that tries to compute the unique intersection point of two lines. This function is not defined for two lines that never intersect nor for two lines that are the same. The Pro/E constraint list $[\texttt{PntOnEnt}^P(id_1^P,\ L_1^P,\ P^P),\ \texttt{PntOnEnt}^P(id_2^P,\ L_2^P,\ P^P)]$ specifies that Pro/E point P^P should be somewhere on line L_1^P and somewhere on line L_2^P. Hence, P^P should be at the intersection of lines L_1^P and L_2^P. If L_1^P and L_2^P intersect at the unique Pro/E point P_{int}^P, then P_{int}^P and P^P should be coincident. So in the SW 2D section, we create the constraint $\texttt{Coincident}^S(conv_O(P_{int}^P),\ conv_O(P^P))$ constraining P^P and the intersection point P_{int}^P to be coincident.

3.5 Extending The Conversion

The programming language approach described in this paper can be extended to reason about converting CAD elements not described in this paper. For example, splines are common entities that occur in CAD systems but are not modeled in *ProLang* and *SWLang*. The approach to modeling Pro/E splines in *ProLang* would start by investigating their internal representation. One then could extend the *ProLang* syntax with a term modeling splines corresponding to the internal representation such as $\texttt{Spline}^P(id^P,\ \overline{P^P},\ r_s,\ r_e)$, where $\overline{P^P}$ is a list of points on the spline curve and r_s and r_e specify the start and end tangent angles of the spline. To preserve the property of being able to create a Pro/E model using only the *ProLang* representation, the internal representation of splines and the associated *ProLang* term should be nearly isomorphic to allow two-way conversion between the formats. Analogously, the *SWLang* syntax can be extended with terms to model splines as they occur in SW. The last step is to extend the conversion semantics with rules defining how to convert between *ProLang* and *SWLang* terms representing splines. Extending our approach to handle other CAD system features would proceed similarly. In most cases, this would be labor-intensive but not a conceptually complex process.

We focused on converting sections from Pro/E to SW. We could have described the conversion in the other direction, i.e, converting from SW to Pro/E, but the approach is similar. Describing conversion in this direction would have greatly lengthened the paper without providing significant conceptual gain.

4 IMPLEMENTATION

To demonstrate that our approach can be fully automated, we implemented the software package, `Pro2DToSW2D`. Given an input Pro/E part file storing a 2D section that can be described in *ProLang*, `Pro2DToSW2D` can automatically create a SW part file containing an equivalent 2D section. `Pro2DToSW2D` is implemented in a modular fashion. It is composed of three software packages that can be executed independently of each other:

1. The `ProE-XMLExporter` application takes in a Pro/E part file and exports an XML representation of the part file.
2. The `ProE-SW-Middleware` application converts an XML representation of a Pro/E 2D section to an XML representation of the equivalent SW section.
3. The `SW-XMLImporter` application takes in an XML representation of a SW 2D section and creates a native SW part file.

The workflow of the conversion process is shown in Figure 8. `Pro2DToSW2D` is just the driver coordinating these applications to execute the workflow for converting the Pro/E native (proprietary) format to the SW native format.

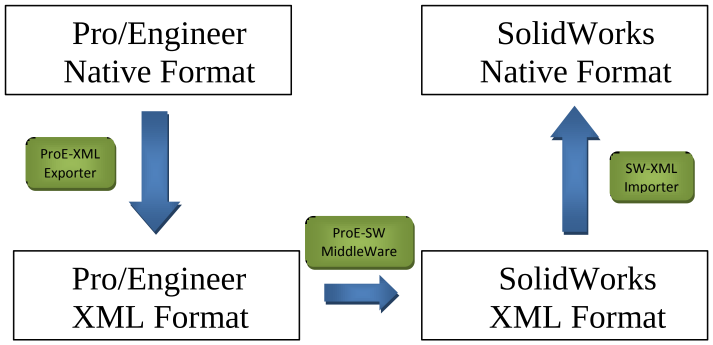

FIGURE 8. Workflow for converting Pro/E 2D sections to SW2D sections

XML [6] is a standard text format that is easy to parse from most popular programming languages. `ProE-XMLExporter` is written in C++ to utilize Pro/E's C++ API, Pro/TOOLKIT [7], for extracting information from Pro/E part files. `SW-XMLImporter` is written in Visual Basic .NET and uses SW's .NET API [8] to write SW part files. The XML intermediate format allows one to choose the best programming language for interacting with the API for each CAD system.

Our `ProE-SW-Middleware` is written in Scala [9]. It reads in an XML representation of a Pro/E section and creates an internal *ProLang* AST representing the section using the grammar given in Section 3.1. It then converts the *ProLang* representation to a *SWLang* representation (as defined by the grammar in Section 3.2) by applying the conversion semantics rules outlined in Section 3.4. The *SWLang* representation is then serialized to an XML representation.

4.1 XML Schemas for Pro/E and SW

Figure 9 shows an XML representation of a segment of a Pro/E 2D section. The root element `<pro2dsection name="S2D0001">` contains a name attribute specifying the name of the section. Within this element are the `<pro2dEntities>`, `<pro2dConstraints>`, and `<pro2dDimensions>` elements containing lists of XML elements for entities, constraints, and dimensions, respectively. The XML tree with root element `<pro2dConstraint id="3" type="PRO_CONSTRAINT_SAME_POINT">` represents a `SamePoint`^{P} constraint. The tree contains XML elements for *entity references*. Entity references allow constraints and dimensions to refer to the entities or parameters of entities they apply to. For example, this `SamePoint`^{P} constraint refers to the first end point of the line with `id=0` using the first entity reference element, `<entityReference id ="0" />`, and the first point type element, `<pointType type ="PRO_ENT_START" />`, in the XML tree of the `SamePoint`^{P} constraint. Point types convey what part of the entity to select. *ProLang* does not contain entity references. Instead when the `ProE-XMLExporter` creates the *ProLang* representation of a constraint or dimension, it just creates a subtree representing the part of the entity referred to by the (entity reference, point type) pair. The *ProLang* AST corresponding to the same point constraint in the XML of Figure 9 is shown below.

`SamePoint`^{P}`(3, Point`^{P}`(0, 0, 0), Point`^{P}`(30, 20, 2))`

The XML schema for SW also contains an XML element for each entity, constraint, and dimension. This schema is similar to the Pro/E XML schema, so we skip its presentation for brevity.

```
<pro2dsection name="S2D0001">
 <pro2dEntities>
  <pro2dEntity id="0" isProjection="true"
    type="PRO_2D_LINE" >
   <end1>
     <Pro2dPnt x="0.00" y="0.00" />
   </end1>
   <end2>
     <Pro2dPnt x="0.00" y="-100.00" />
   </end2>
  </pro2dEntity>
  <pro2dEntity id="2" isProjection="false"
    type="PRO_2D_POINT">
   <pnt><Pro2dPnt x="30.00" y="20.00" /></pnt>
  </pro2dEntity>
 <pro2dConstraints>
  <pro2dConstraint id="3"
    type="PRO_CONSTRAINT_SAME_POINT">
   <entityReferences>
    <entityReference id ="0" />
    <entityReference id ="2" />
   </entityReferences>
   <pointTypes>
    <pointType type ="PRO_ENT_START" />
    <pointType type ="PRO_ENT_WHOLE" />
   </pointTypes>
  </pro2dConstraint>
 <pro2dDimensions>
  <pro2dDimension id="5" type="PRO_TK_DIM_LINE"
    value="100.00" >
   <entityReferences>
     <entityReference id ="0" />
  </entityReferences>
   <pointTypes>
     <pointType type ="PRO_ENT_WHOLE" />
    </pointTypes>
  </pro2dDimension>
 </pro2dDimensions>
</pro2dsection>
```

FIGURE 9. Example XML Representation of a Pro/E 2D section

5 CONVERTED SECTIONS

This section presents two example Pro/E sections that our software converted. First, we present a Pro/E 2D section converted by our software that requires converting a combination of multiple Pro/E elements to a combination of multiple SW elements. This Pro/E section contains a `PntOntEnt`^{P} constraint. *SWLang* has no equivalent counterpart to `PntOntEnt`^{P} constraints. The example Pro/E section is shown in Figure 10 with the entities, constraints, and dimensions labeled.

An interesting case of non 1-1 mapping arises when constraining the corner of a rectangle to the origin. In Pro/E, the origin is marked by the intersection of the x and y axes. These axes are listed as entities and have identifiers 0 and 1. The `PntOntEnt`^{P} constraint with `id="8"` constrains the first end point of the bottom line of the rectangle, the `Line`^{P} entity with `id="4"`, to the x axis. Also, this point is constrained to the y axis by the `PntOntEnt`^{P} constraint with `id="9"`. Since the axes intersect at the origin, the point must be located at the origin.

`ProE-SW-Middleware` read in the XML representation of the Pro/E XML section and created an internal *ProLang* AST. Then it converted the *ProLang* representation to a *SWLang* representation by applying the conversion semantics rules. The resulting SW section is shown in Figure 11. Note the `Coincident`^{S} constraint constraining the start point of the bottom line of the rectangle, the `Point`^{S} entity with `id=(0,1)`, to be coincident with the point at `(0,0)`. Hence, rule `INTSECT` was applied to convert this model.

Figure 12 shows an example Pro/E section only containing entities, constraints, and dimensions that can be converted in a 1-1 manner to SW 2D elements. We converted this Pro/E section to the SW section shown in Figure 13. The `LinePointDim`^{P} dimension between the Pro/E y axis and the left side of the rectangle was converted to a `HorLineDim`^{S} between the origin and the point at the bottom-left corner of the rectangle.

6 RELATED WORK

This section compares our programming language approach to other popular CAD data exchange approaches. IGES [10] and STEP [11] are neutral, geometric-based formats that are

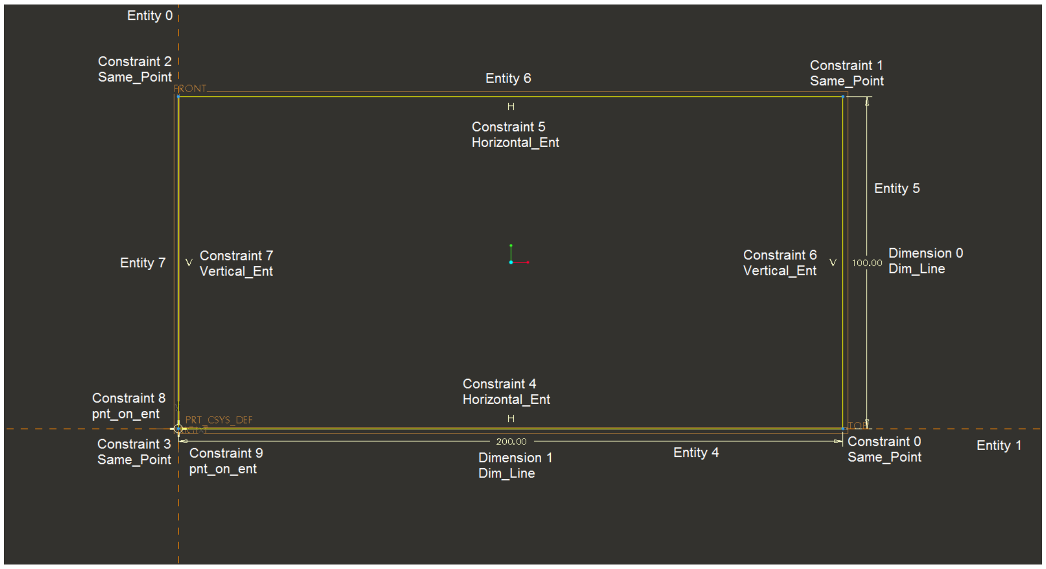

FIGURE 10. *ProLang* 2D section of rectangle constrained on origin

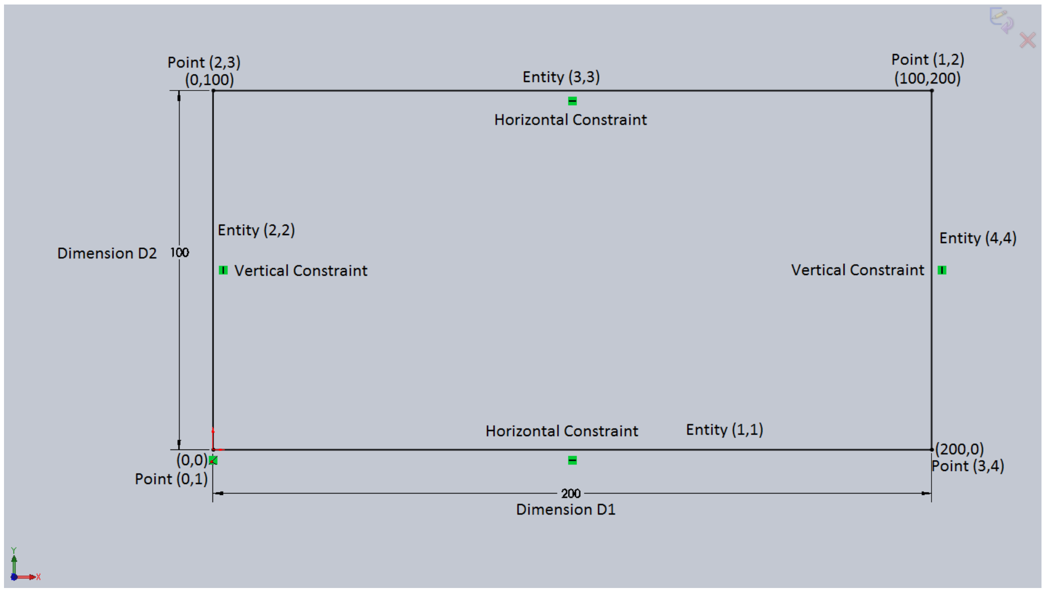

FIGURE 11. Output SW 2D section from converting Pro/E section in Figure 10

commonly used to exchange geometric data between CAD systems. Although they enable transferring geometric data accurately between CAD systems, they do not preserve parametric-based CAD data such as construction history, features, and constraints. Hence, these formats are inadequate for modifying, extending, and performing other important higher-level functionality on CAD models in a target CAD system, where the models were created in a different source CAD system. For example, Figure 14 shows a Pro/E model and its feature tree when opening its native part file in Pro/E. Figure 15 shows the same Pro/E model and its feature tree when opening an IGES representation of the model from SW. The features from the native part file have been lost. Our approach enables all design-intent, not just geometric information, to be exchanged between CAD systems.

Proficiency [12] is a commercial CAD interoperability application that converts between Pro/E, CATIA V4 and V5, NX,

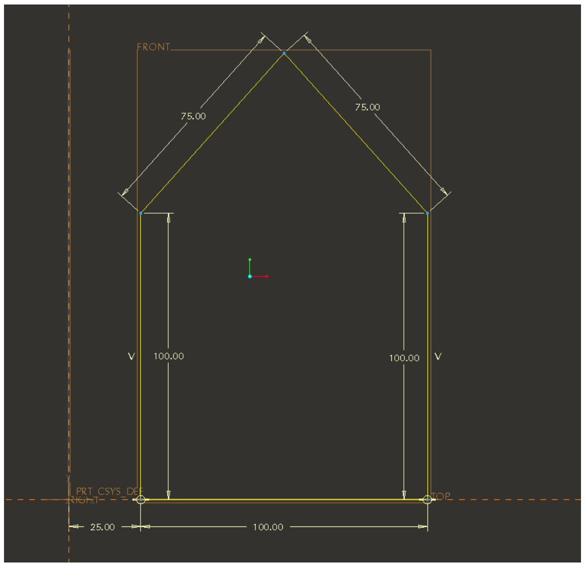

FIGURE 12. *ProLang* 2D section with lines not perpendicular or parallel to each other

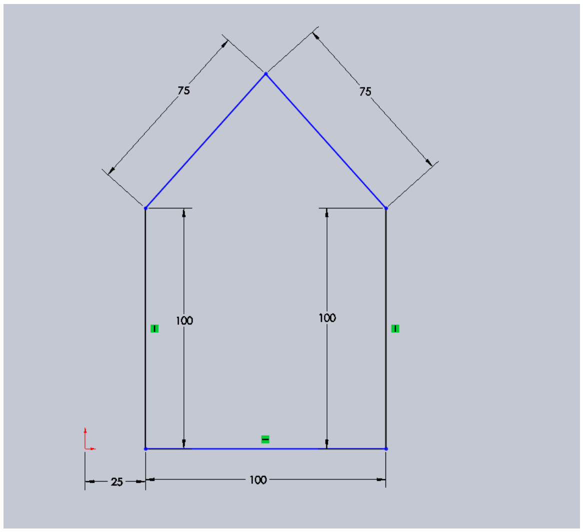

FIGURE 13. Output SW 2D section from converting Pro/E section in Figure 12

and I-deas. Native CAD formats are converted to Proficiency's closed, binary, and proprietary *Universal Product Representation* [13] (UPR). Proficiency is able to automatically transfer feature-based information between CAD systems when the features map in a 1-1 or 1-many manner. Features which do not map 1-1 or 1-many are converted using Proficiency's Completion Wizard, which prompts the user to decide how features should be converted. Either the user converts the feature to a solid ge-

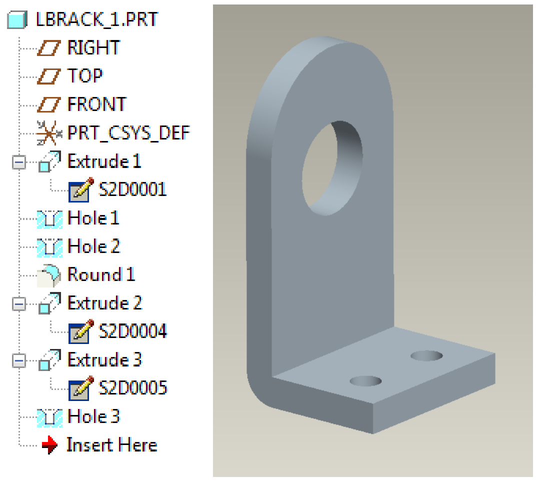

FIGURE 14. Feature tree for a native Pro/E part file

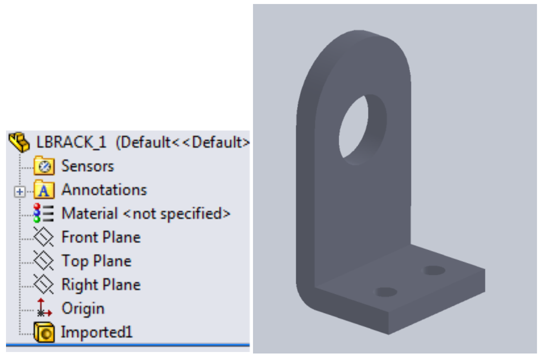

FIGURE 15. Feature tree for the same part in Figure 14, except that it has been saved in IGES format

ometric object, or the user manually tries to recreate the feature in the target CAD system. Because their UPR representation is a closed, binary format, the semantics of their information and the parametric data that can be represented in UPR is not readily apparent to a user. In addition, Proficiency does not support automated conversion between many-to-many or many-to-one mappings.

6.1 Comparison To Ontological Approaches

Ontologies [14] have arguably become the dominant approach to parametric CAD data exchange [15, 16, 17, 18, 19]. The following is an overview of the ontological-based approach. Each CAD system is represented as an ontology containing classes/concepts, properties, and constraints. CAD models are represented by a graph [20], where the instances of the classes in the ontology are the nodes of the graph and the property relations between these instances are the edges of the graph. For example, [15] presents PRO-AO, the Pro/E Application Ontology, and CAT-AO, the CATIA Application Ontology, both described in the OWL Web Ontology Language [21]. In [16] the authors describe how they compute similarity measures between classes from their ontologies. Although the authors describe how they map instances with similarity metrics between classes, they do not explain how they *convert* instance data from a source ontology to instance data in a target ontology. Since CAD models are represented using instance data, they never define how to convert CAD models from one system to another.

Even given equivalences between OWL classes, important unanswered questions remain for using OWL to convert CAD models. One approach to using equivalence relations between OWL classes is based on the fact if I is an instance of class C and class C' is equivalent to C, then OWL will automatically infer that I is also an instance of C'. Class C can be defined in a source ontology, and C' can be defined in a target ontology. However, inferring that I is an instance of class C' in the target ontology does not specify how to convert between the parameters required for the classes.

Consider the following scenario. CAD system A defines a feature type `Plane3Points` that takes in three points as parameters to define a plane. CAD system B defines a feature type `PlaneNormal` that takes in one point and a normal vector as parameters to define a plane. An ontology for A contains a class named `Plane3Points`, with property restrictions specifying that instances of this class require three distinct points. An ontology for B contains a class named `PlaneNormal`, with property restrictions specifying that instances of this class require one point and a normal vector. Let P_1, P_2, and P_3 be three distinct points, which define a plane. Let P and N be the point and normal vector that define the same plane as P_1, P_2, and P_3. Let I be an instance of class `Plane3Points` that represents the plane containing points P_1, P_2, and P_3. Since I is an instance of class `Plane3Points`, there must exist instances that represent points P_1, P_2, and P_3, and these instances are property values of I. However, there is no requirement that instances for point P and vector N exists. If classes `Plane3Points` and `PlaneNormal` were specified to be equivalent, how would the instances for P and N be generated? One may consider defining SWRL [22] rules to compute P and N, but SWRL rules can only infer new property relationships between existing OWL individuals, classes, or other resources. [1] In addition, to the best of our knowledge, no one has shown how to compute the complex mathematics required in the logic to convert between CAD formats within OWL, SWRL, or the semantic web. Hence, automatically generating instances for P and N would have to be performed by a separate application that carries out the mathematical computation to determine P and N and generate the corresponding instance data in the OWL format. Our approach allows such conversions to be computed over programming languages representing CAD systems.

We are not aware of any implementations of ontology-based approaches capable of actually converting between proprietary CAD formats, nor even converting from an ontological representation to a proprietary CAD format. An advantage of our approach is that our languages are linked with the semantics of the CAD system by software. One can create a Pro/E model in the native Pro/E *without using the Pro/E GUI* by writing a program in *ProLang* in the XML format specified in Section 4.1. Similarly, one create a SW model by only writing XML. However, to our knowledge, no existing ontological format allows one to a create a CAD model in a native CAD system format by only specifying the model in the ontological format.

An intermediate format that is not paired with software that translates the intermediate format to a format that a CAD system can understand cannot be readily compared to the semantics of the CAD system it is attempting to model. As a result, it is not clear how close the ontological representations of CAD models are to the proprietary format used by real CAD systems. Consider the feature formats from [1] and [17]. Both formats are used to create feature representations of SW models. One difference between the two formats is the format from [17] has a concept of a *Form* but the format from [1] does not. The differences between the two formats implies there is a semantic difference between at least one of these formats and the proprietary format used by SW. Lacking a way to connect them to real CAD systems makes it difficult to determine what makes the ontological format for representing SW models from [17] better than the format from [1] or even the pizza ontology [24]. A concrete link is needed to explain the correspondence between an intermediate representation and the format used by a CAD system for its models such as software that translates between the intermediate format and the native CAD format. Our prototype system includes such a link.

Lastly, our formalism allows more compact representations of CAD systems and models than OWL, and the logic of the conversion process is more apparent. For example, relationships between $n \geq 3$ elements or n-ary relations are compactly represented with terms such as $\texttt{LineDim}^S(idd,\ r,\ E^S_1,\ E^S_2)$. However, an OWL property relation can only be between two nodes, and representing n-ary relations in OWL is much more cumbersome requiring one to understand more details about the semantic web. Because of the complexity of representing n-ary relations in OWL, W3C has created a lengthy and detailed document explaining how to do so [25].

[1]The Protégé SWRLX [23] extension to SWRL does allow creating new OWL resources in inference rules, but this extension is not a W3C standard and a Protégé specific plugin.

7 SUMMARY

We have presented a novel approach to parametric CAD data exchange based on formal foundations using techniques from programming language research. We have demonstrated that our approach can be automated to convert models between popular CAD systems. Our software can enable other CAD interoperability researchers to automate their approaches using our open, easy-to-parse XML formats without learning CAD APIs. We have created programming languages modeling (subsets of) Pro/ENGINEER and SolidWorks, *ProLang* and *SWLang*. We have rigorously described our algorithm for converting parametric CAD data between the modeled subsets of Pro/ENGINEER and SolidWorks by defining conversion semantics that converted *ProLang* trees to *SWLang* trees. Our approach can convert any section that can be represented in *ProLang* that is the union of subsections that have equivalent SolidWorks sections. Although the proof-of-concept case study presented in this paper only models a subset of 2D sections, our approach is not limited to 2D data. Features and other 3D data from a source CAD system can equally well be represented by terms from a language and be converted for use in a target CAD system by defining conversion semantics to reason about its translation.

ACKNOWLEDGMENT

This work was supported by the NSF Center for e-Design (NSF Award No. 0838747 to the University of Massachusetts).

REFERENCES

[1] Hanayneh, L., Wang, Y., Wang, Y., Wileden, J., and Qureshi, K., 2008. "Feature Mapping Automation for CAD Data Exchange". *2008 ASME International Design Engineering Technical Conferences & The Computer and Information in Engineering Conference (IDETC/CIE2008)*(DETC2008-49671).

[2] Chen, X., and Hoffmann, C., 1995. "On Editability of Feature-based Design". *Comp.-Aided Des.,* ***27***(12), pp. 905–914.

[3] Hoffmann, C., 1997. *CAD Systems Development*. Springer,Berlin, ch. EREP Project Overview, pp. 32–40.

[4] Shih, C., and Anderson, B., 1997. "A Design/Constraint Model to Capture Design Intent". *Proc. 4th ACM Symp. on Solid Modeling & Applications,* ***27***(12), pp. 255–264.

[5] National Institute of Standards and Technology. `http://www.nist.gov/sc4/paramet/short/engen/edm46.pdf`.

[6] W3C. XML. `http://www.w3.org/XML/`.

[7] PTC Logo Creo Elements/Pro TOOLKIT Customization API. `http://www.ptc.com/products/creo-elements-pro/protoolkit-customization-api`.

[8] SolidWorks API Help. `http://help.solidworks.com/2010/English/api/sldworksapiprogguide/Welcome.htm`.

[9] Scala. `http://www.scala-lang.org/`.

[10] Initial Graphics Exchange Specification. `http://ts.nist.gov/standards/iges/`.

[11] WikiStep. `http://www.wikistep.org/`.

[12] Proficiency. `http://www.transcendata.com/products/proficiency/`.

[13] Spitz, S., and Rappoport, A., 2004. "Integrated feature-based and geometric cad data exchange". In Proceedings of the ninth ACM symposium on Solid modeling and applications, SM '04, Eurographics Association, pp. 183–190.

[14] Ontology Development 101. `http://protege.stanford.edu/publications/ontology_development/ontology101.html`.

[15] Zhu, L., Jayaram, U., Jayaram, S., and Kim, O., 2009. "Ontology-driven integration of cad/cae applications: Strategies and comparisons". *ASME Conference Proceedings,* ***2009***(48999), pp. 1461–1472.

[16] Kim, O., Jayaram, U., Jayaram, S., and Zhu, L., 2009. "An ontology mapping application using a shared ontology approach and a bridge ontology". *ASME Conference Proceedings,* ***2009***(48999), pp. 431–441.

[17] Patil, L., Dutta, D., and R., S., 2005. "Ontology-based exchange of product data semantics". *IEEE Transactions on Automation Science and Engineering,* **2**, July, pp. 213–225.

[18] Zhan, P., Jayaram, U., Kim, O., and Zhu, L., 2010. "Knowledge representation and ontology mapping methods for product data in engineering applications". *Journal of Computing and Information Science in Engineering,* ***10***(2), p. 021004.

[19] Wang, Y., and Nnaji, B., 2006. "Document-Driven Design for Distributed CAD Services in Service-Oriented Architecture". *ASME Journal of Computing and Information Science in Engineering,* ***6***(2), pp. 127–138.

[20] Graph theory. `http://en.wikipedia.org/wiki/Graph_theory`.

[21] W3C. OWL Web Ontology Language Guide. `http://www.w3.org/TR/owl-guide/`.

[22] W3C. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. `http://www.w3.org/Submission/SWRL/`.

[23] Protégé SWRL Extensions BuiltIns. `http://protege.cim3.net/cgi-bin/wiki.pl?SWRLExtensionsBuiltIns`.

[24] Pizza Ontology. `http://www.co-ode.org/ontologies/pizza/`.

[25] W3C. Defining N-ary Relations on the Semantic Web. `http://www.w3.org/TR/swbp-n-aryRelations/`.