Category Theory-Based Synthesis of a Higher-Level Fusion Algorithm: An Example

Mieczyslaw M. Kokar
Electrical & Computer Eng.
Northeastern University
360 Huntington Avenue
Boston, MA 02115, USA
mkokar@ece.neu.edu

Kenneth Baclawski
Computer Science
Northeastern University
360 Huntington Avenue
Boston, MA 02115, USA
kenb@ccs.neu.edu

Hongge Gao
Electrical & Computer Eng.
Northeastern University
360 Huntington Avenue
Boston, MA 02115, USA
hgao@ece.neu.edu

Abstract - ***Higher-level fusion (e.g., Level 2 that deals with derivation of relations among objects) often involves symbolic processing of information obtained from lower levels (e.g., Level 1, which deals with object detection, identification and tracking) which is based upon quantitative algorithms. The quantitative algorithms pass only some information to higher levels; some of the information is abstracted away in this process. However, this information might be needed by some of the rules at a higher level. Should in such a case the whole system be re-coded? In this paper we present an attempt to overcome the need for re-coding the Level 1 software. Instead, we propose to replace the activity of manually developing a fusion algorithm with an automated synthesis of the specification of the algorithm followed by automatic code generation. While solving such a problem in its entirety is a rather distant goal, in this paper we propose a solution to a more modest sub-problem. Rather than attempting to solve arbitrary information fusion problems, we assume that there exists a library of templates that specify information fusion objectives. The templates are formal specifications represented in a formal language. Since they are declarative, a variety of algorithms can satisfy their requirements. This paper presents an example of using a formal, category theory-based approach to the problem of synthesizing algorithms that satisfy templates for information fusion.***

Keywords: Information fusion, category theory, system synthesis, templates.

1 Introduction

One of the definitions of information fusion (or data fusion) [1] is: "Data fusion is the process of combining data or information to estimate or predict entity states." The JDL model [1] associates entity types with "levels". For instance, Level 1 deals with objects, e.g., aircraft or tank. Level 2, on the other hand, deals with relations among such objects. Relations are assessed by some algorithms (or rules) that take various features of the objects detected at Level 1 and use some background knowledge to decide whether a relation among given objects holds or not. Since the number of possible relations among the objects is exponentially high, it is rather impossible to hard-code all of the algorithms that might be needed to analyze a specific type of relation. To address this problem, ontology-based approaches have been proposed[1]. The main idea behind ontology-based processing is that possible types of relations are specified by an ontology, and a generic logic-based inference engine is used to derive any such relation (rather than hard-coding an algorithm for each relation). While the use of a logic-based inference engine for relation derivation may be reasonable for Level 2 processing, where a lot of information is symbolic, this does not seem to be a practical solution for Level 1, where most of the processing is quantitative. In symbolic processing, an algorithm is, in a sense, synthesized dynamically. This is actually accomplished by the generic mechanism of the inference engine. Since quantitative algorithms are hard-coded, one cannot achieve the flexibility comparable to the symbolic processing at Level 2. Unfortunately, this also leads to some problems, as described below.

Consider a scenario in which an analyst wants to observe a specific type of relation. Suppose the Level 2 knowledge base has a rule for deriving such a relation, but the rule requires some information that Level 1 processes use in their own processing but do not provide to the higher levels. One way to deal with this would be to rewrite the Level 1 processes so that they pass more information to Level 2. The disadvantage of this solution is that this Level 1 process would be passing more information every time it is invoked, even though the additional information might be needed only occasionally. An even bigger problem with this solution would be that the whole system would have to be re-

[1] One of the definitions of the term "ontology" is given in [2]: "An ontology is a specification of a conceptualization."

programmed, re-compiled and re-tested whenever such a new request by the analyst was posted.

In this paper we describe our efforts to solve the above problem in a different way. Our goal is to investigate an approach that is based on dynamic software composition, or more specifically on Model Driven Architecture (or MDA) [3]. In the MDA approach, it is assumed that software is specified using UML models (i.e., diagrams). In our case, we use formal specifications for software. Such a specification (also referred to as a *spec*) contains some information about each of the variables and each of the functions of the software. All specs are represented in a language with mathematically defined and computer executable semantics. Similarly, the rules at the higher level are also represented in the formal language. Consequently, the rules and the specs can be naturally integrated within a formal deductive system. In other words, the problem of dynamic execution of a rule at Level 2, which in turn requires information from Level 1, is equivalent to synthesizing an algorithm that integrates the Level 1 specs with the Level 2 rules. While it would be desirable to have the whole synthesis task done in a fully automatic way, in this paper we present a more modest approach in which the synthesis process is constrained by a template selected by the analyst.

In the next section we briefly overview the software development process. The main intent here is to identify the activities that can be assigned to two different actors in this process: the fusion expert and the software developer. Section 3 gives a brief overview of the basics of the formal method approach to the development of software. Section 4 introduces the basic concepts of category theory. Our approach is based on category theory in that it treats specifications of software modules as objects in a category and then uses the category theory operation of *colimit* as a mechanism for combining multiple specifications into one. To show the steps involved in the composition of a higher-level fusion module we use an example of an intrusion detection system able to recognize one type of intrusion - the *smurf attack* [4]. This example is discussed in Section 5. In Section 6 we describe the sensors used in the intrusion detection process. Then we show a *template* that can be used for generating queries about smurf attacks. This is followed by the step of mapping the problem into the category theory framework (Section 8). In Section 9 we describe the code generation process and the resulting code. And finally in Section 10 we present conclusions and future directions for this line of research.

2 Software Development

A large portion of the effort of developing an information fusion system is the development of the software to perform the information combination (fusion) operations. This includes such engineering activities as system specification, software specification, software design, algorithm development, software implementation, testing and maintenance. Scientists in the information fusion domain deal mainly with algorithm development. The fusion algorithms must (typically) perform such functions as data association, object detection, recognition and tracking, relation derivation, situation assessment, and so on. While there is always a prerequisite that there must be multiple sources of information involved, the methods of analysis of that information are not particularly specific to the field of information fusion. As often stated in the fusion literature (cf. [5]) the algorithms are standard algorithms for computing the uncertainty of particular decisions rather than being specific to fusion.

The above listed activities of fusion system development require at least two kinds of expertise - information fusion and software engineering. The two kinds of expertise are rarely found in one person. Therefore it would be good to have a clear partition of the task of fusion system development into subtasks each of which requires just one kind of expertise. Furthermore, one could also try to automate as much of the development effort as possible. One condition that must be satisfied, though, is that the interface between a fusion task and the software engineering task should be specified very precisely so that there is no confusion about the semantics of the dependency between the two tasks. One way to achieve such a clarity is to use the formal method approach of software development [6, 7, 8, 9].

The automation of specification development is a very difficult task. Typically the specification of a system is captured in text and possibly some pictures. Then this representation is coded into a formal method system. Obviously, it would be nice if we could have a translator that could read the text and then output a specification in a formal method language. Unfortunately, such a situation is a distant, perhaps even unachievable, goal. In this paper we discuss an approach in which a specification developer first specifies signatures of three kinds of processing module: *sensor modules, domain modules* and *goal modules*. Sensor modules capture the functions of particular sensors. In short, such a function assigns some sensor values (data) to sensor coordinates. Domain modules capture the functions that assign *object values* such as object names (labels) to

world coordinates. And finally, a goal module captures the problem (the goal) that a given fusion system is supposed to solve. In general, any fusion system must assign some object values to world coordinates based upon sensory (or, more general, data) inputs. In the second step, the specifier browses through a library of templates of fusion systems to find one that matches the sensor, domain and goal modules. The next step is to find and include specifications from a library of specifications relevant to the templates. We assume that such a library is available; we call it the *background knowledge specification library*. This step can be supported by a computer system. Finally, the last step is to perform code generation.

3 Formal Methods

The use of formal methods in software engineering is a very active research area. The applicability of these methods is currently limited mainly to *safety-critical systems* [7]. Nevertheless, there is an expectation that the use of formal methods in software development will eventually be not only more popular, but also less expensive than the traditional approaches to software development [10].

Using the formal method approach, one first specifies a system formally and then develops the code through the process called *refinement* [11]. Such a two-step approach gives a nice partition of responsibilities - specifications can be developed by fusion experts and the code development can be the responsibility of a software engineer. This also means that the goal of automation of development of a fusion system can be split into the automation of specification development and automation of refinement.

Note that combining algorithms is a more complex operation than just combining data. Data can be combined using various set theoretic operations, such as union, intersection, Cartesian product, and operations for combining uncertainty. Such operations cannot be used directly to combine algorithms, and other kinds of combination operations are needed. Category theory [12] is a framework which can combine algorithms, using operations such as the colimit which is introduced below.

If one uses the category-based Specware tool [13], the latter step can be automated to a great degree essentially for free, because Specware supports code generation. We used this feature in our research reported in this paper.

4 Category Theory

A *category* [12] is a mathematical structure consisting of *category objects*, *category arrows* (or *morphisms*) and a *composition* operation on certain pairs of morphisms such that composition is associative. For instance, in category `Set` the objects are all sets. Category arrows define the relationships between pairs of objects, with the first object called the *domain* and the second called the *codomain*. For example, if P and Q are category objects, then an arrow f with domain P and codomain Q would be written $P \xrightarrow{f} Q$. A pair of arrows f and g is *composable* if the codomain of f is the domain of g. For instance, in the category `Set` the arrows are functions between sets. For every pair of composable arrows f and g the composition operation assigns an arrow $g \circ f$ whose domain is the domain of f and whose codomain is the codomain of g. In the category `Set` the composition is the composition of functions. A *diagram* in a category is a collection of objects and a collection of arrows between these objects. A diagram is *commutative* if the composition of arrows within the diagram is always consistent. For example, the following diagram of four objects and four arrows

$$\begin{array}{ccc} P & \xrightarrow{p} & Q \\ {\scriptstyle r}\downarrow & & \downarrow{\scriptstyle q} \\ R & \xrightarrow[s]{} & S \end{array}$$

is commutative provided that the composition of the arrows p and q is the same as the composition of the arrows r and s, i.e., $q \circ p = s \circ r$.

The *colimit* of a diagram D is the object that "fuses" the objects of the diagram while unifying the common (*shared*) parts of the diagram. More precisely a colimit is an object C with the following two properties:

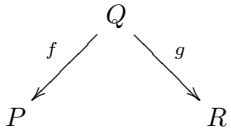
1. There is a morphism from every object in D to the object C such that these morphisms together with the morphisms in D define a commutative diagram. These morphisms are called the *cone*, with C as the *apex* of the cone.

2. If B is any object that satisfies the first property, then there is a unique morphism from C to B that commutes with the cones of B and C.

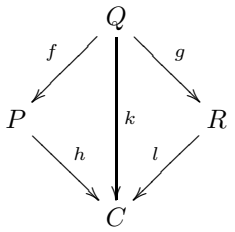
The first property ensures that C fuses all of objects of the diagram. However this property alone would allow C to have additional parts or to fuse parts that are not in common. The second property ensures that C exactly fuses the objects of the diagram and unifies only the common parts and no others. The second property implies that a colimit is unique up to isomorphism, and for this reason one usually refers to it as *the* colimit. For some categories the colimit of a commutative diagram may not always

exist. Accordingly, for each category one must check that colimits exist.

For example, in the category Set, let P=$\{1,2,3\}$, R=$\{6,7,8\}$, and Q=$\{3\}$, where the numbers represent identifiers of objects in two views of a scene. Next define the morphisms (functions) f and g on Q that $f(3)$=3 and $g(3)$=7. The colimit of the commutative diagram



is a 5-element set. For example, C=$\{1,2,3,6,8\}$ is a colimit. By definition of the colimit, there is a commutative diagram of morphisms from P, Q, and R to C as follows:



The morphisms h, k and l map the elements of P, Q and R to C exactly as one would expect, except that $l(7)$=3. Intuitively, C is obtained by combining (fusing) the elements of P and R so that 3 and 7 identify the same object in D. C is the set of common elements of P and R.

For specifying fusion systems we use the category Spec. The objects of Spec are algebraic specifications (or *specs* for short) which consist of three kinds of element:

1. A *sort*, also called a *type*, is a set of values. Sorts can be constructed from other sorts of the spec by using constructs such as the Cartesian product or the exponential (i.e., the set of all functions from one sort to another).

2. An *operation* or *op* is a function from one sort to another sort. Like sorts, ops can be defined in terms of other ops of the spec.

3. An *axiom* is a requirement or constraint that the sorts and ops must satisfy.

An *interpretation* of a spec is a choice of a set for each sort and a choice of a function for each operation such that the functions have the appropriate signatures and the axioms are all true statements about the sets and functions. A *theorem* of a spec is a statement that is true in any interpretation of the spec. A spec is *consistent* if it has at least one interpretation. We always assume that our specs are consistent.

A morphism in the Spec category maps sorts and operations of a domain spec to sorts and operations of a codomain spec such that axioms of the domain spec are theorems in the codomain spec. Goguen showed that Spec has the property that every finite diagram has a colimit [14]. The colimit operation creates a new specification from a diagram of existing specifications. This new specification has all the sorts and operations of the original set of specifications without duplicating the shared sorts and operations.

For a more complete treatment of category theory see [12]. Algebraic specifications are described in [16]. Examples related to information fusion can be found in [17, 18, 19, 20].

5 Smurf Attack Detection

To demonstrate the idea of template-based specification synthesis we use an example of intrusion attack detection. In particular, we focus on one type of attack called the *smurf attack*. The purpose of such an attack is *denial of service*. It may be very harmful since the computers on the network will not be able to communicate, or even operate, due to the high traffic congestion and workload generated by the attack.

The way this attack works is that the attacker *broadcasts* (i.e., sends to a whole subnetwork) an *echo request* message. According to the protocol, all the computers that receive such a request are supposed to reply with an *echo reply* to the sender. However, the attacker *spoofs* the sender's address, i.e., it uses the address of another machine. Consequently, the machine whose address was selected by the attacker (the *target* of the attack) starts receiving high volumes of echo reply messages. The attacker may amplify this effect by sending echo request messages repetitively to several subnets. The result is that not only the target machine is saturated with receiving echo replies, but also whole subnets are saturated with sending echo replies.

While detecting such a form of attack seems to be easy, the handling of the action must be done rather carefully. The echo request message is a legitimate way of establishing communication with other machines. Care must be taken to avoid *false positives*, i.e., declaring that a specific user is an intruder. A single echo request message or even a number of them in rapid succession is not sufficient evidence that a smurf attack is taking place. Similarly broadcasts are also not, by themselves, sufficient evidence for a smurf attack. Any of these conditions could occur during normal network operations. Information of several kinds must be combined before a smurf attack can be reliably detected.

A more sophisticated version of this attack would involve a number of collaborating intruders. In such a case each of the intruders could send an echo request to a subnet infrequently.

This would not look suspicious to an intrusion detection program. While there are ways of handling this kind of attack, in this paper we deal only with the simple version.

6 Sensors

To detect an attack the network must be monitored using some form of sensor. For network attack detection, the sensors are processes that run in "promiscuous" mode, i.e., they look inside of the headers of all frames of information being transmitted by a network. The headers contain information about the communication, such as the source address, the destination address, the time stamp, the size of the content in the frame, and so on. The role of a specific sensor is to extract (project) some information of interest from the headers.

In the experiments described in this paper we used the *tcpdump* program to capture header information, and we used two simple sensors based on tcpdump. One of the sensors extracts the source address and the second one extracts the ICMP message type. There are many message types, including echo request, echo reply, time exceeded, and destination unreachable. The following is small part of a filtered tcpdump trace:

```
00:00:05.327 10.1.101.7 > 192.168.15.255 8
00:00:05.327 10.1.101.7 > 192.168.1.255 8
00:00:05.327 10.1.101.7 > 192.168.15.255 8
00:00:05.327 10.1.101.7 > 192.168.1.255 8
```

Each line of this trace shows some information about one instance of communication (one packet). The first part (e.g., `00:00:05`) shows the time stamp, i.e., the time at which the communication took place. This is followed by the port number (in this case it was port `327`). Then the source host address is shown (`10.1.101.7`). After the bracket the target host address is shown (`192.168.15.255`). The last item is the type of request.

7 Template-based Specification

Templates are structures that include two kinds of component: constants and variables. An example of a template is a tax form, in which the constant part is printed and the variable parts, like ones income, need to be filled in. In our case, we use the MetaSlang language to specify both templates and concrete specifications. In MetaSlang one can distinguish *abstract* and *concrete* specifications. A sort is concrete if it is fully defined in terms of interpreted sorts, such as integers and strings. A sort is abstract if it is not concrete. Similarly an operation is concrete if it is fully specified by axioms, and an operation is abstract if it is not concrete, e.g., if only the operation signature is specified. Concrete sorts and operations correspond to the "constant" parts of a form, while abstract sorts and operations are the "variable" parts. Filling in a form replaces the variable parts with constant values. Refinement is a similar process whereby abstract sorts and operations are replaced by compatible concrete ones.

The simplest possible nontrivial spec consists of a single sort with no ops and no axioms. In MetaSlang this is written as follows:

```
T = spec
  sort X
endspec
```

The spec `T` represents a single abstract datatype with no specified operations. The following is an example of spec that has an op:

```
S = spec
  sort A, B
  op f: A -> B
endspec
```

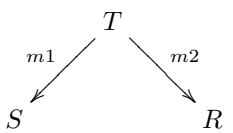
The spec `S` has two abstract sorts `A` and `B`, and a function `f` from `A` to `B`. The sort `S` is an abstraction of a general deterministic computer program, where `A` is the set of possible input values, `B` is the set of possible output values, and `f` is the mapping from inputs to outputs. Axioms added to `S` will specify various required features of the behavior of the program.

One can easily define a morphism from the spec `T` to the spec `S`. In fact, there are two possible morphisms: the sort `X` can correspond to either `A` or `B`. The latter morphism is written in MetaSlang as follows:

```
m1 = morphism T -> S  X +-> B
```

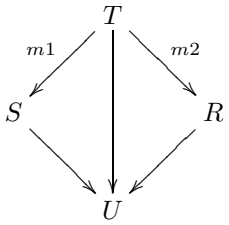
The notation `+->` is used to denote a correspondence. In this case, `X` corresponds to `B`. A distinct notation was introduced to avoid confusing correspondences with functions. In the morphism `m1`, there is no function from `X` to `B`.

One can combine (or *fuse*) formal specifications by using the colimit operation. One can show that every commutative diagram in the category `Spec` has a colimit [14]. As a simple example, suppose that we have a spec `R` that is just like `S` but with sorts `C`, `D` and op `g`. One can think of `R` as another computer program. Suppose one also has a morphism `m2` from `T` to `R` that corresponds `X` with `C`. The two morphisms `m1` and `m2` define a diagram of morphisms as follows:



The fusion of this diagram is a spec U that has three sorts and two ops, and the morphisms to

U are in the following commutative diagram:



The effect of the fusion could be, for instance, to unify the output of the first program with the input of the second program. In other words, one "feeds" the result of the first program into the second program, one of the most common ways to combine programs.

8 Example of Template-based Specification Synthesis

The specification development proposed in this paper is based upon the assumption that a library of templates and diagrams is available for use by the specifier. In addition, we assume that a library of concrete specs (or *background knowledge* specs) exists and can be incorporated into the ultimate specification either manually by the specifier or automatically by a Specification Matcher, which we are developing as part of our research. The specifier first needs to develop a formal specification of the need in terms of an abstract goal and abstract specifications of the sensors.

Specification development consists of the following steps.

1. Find a template that matches both the Sensor specs and the Goal spec.
2. Find concrete specs that match the abstract sensor specs.
3. Refine the abstract template specs.
4. Find concrete specs in the library that need to be included in the final spec.
5. Perform the colimit operation on the refined template diagram.
6. Refine the resulting spec into code.
7. Test the code and improve on any previous steps, if necessary.

For the smurf attack example, we will combine three computer programs using the diagram in Fig. 1 which we assume is available in the library. In this diagram, the specs S_1, S_2, S_{c_1} and S_{c_2} have the structure of an abstract computer program as in Section 7, i.e., each one has two sorts and one op with no axioms. The specs S_1 and S_2 are the abstract sensor specs, while

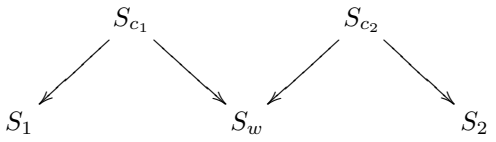


Figure 1: The specification fusion diagram for the smurf attack example

S_{c_1} and S_{c_2} are introduced to enable the proper unification in the fused specification. The spec S_w is the abstract Goal spec, and it has a more complex structure than the others because it is specific to the needs of the particular problem of detecting attacks from sensor information. The following is one possible Goal spec:

```
Sw1 = spec
  sort P, Y, Q1, Q2
  op g: P -> Y
  op s1: P -> Q1
  op s2: P -> Q2
  op ff1: P -> Q1*Q2
  op ff2: Q1*Q2 -> Y
  axiom ffA is fa(p:P) ff1(p) = (s1(p),s2(p))
  axiom gA is fa(p:P) g(p) = ff2(ff1(p))
endspec
```

In this specification, the input sort for the program is written `P`, and the detector program is the op `g`. The sort `Y` represents the possible detection answers, such as "attack detected," "no attack detected," and "possible attack detected." The ops `s1` and `s2` are the sensor programs. The two sensor programs are combined (but not composed) in the op `ff1`. The asterisk is used to denote the Cartesian product. Thus `Q1*Q2` is the set of ordered pairs of elements of `Q1` and `Q2`. Axiom `ffA` defines `ff1` to be the function that maps each input element to a pair consisting of the two sensor reports. Thus `ff1` is a concrete op in this spec, unlike the sensor specs which are just abstract specs. Both sensor specs become concrete only after all the specs are fused using the colimit operation. The op `ff2` is the function that determines whether there is an attack based on the outputs of the two sensors. The goal program `g` is computed by applying `ff2` to the result of `ff1`. This is stated in axiom `gA`.

The problem with the Goal spec `Sw1` above is the presumption that one can detect an attack by using a single sensor output value. In fact, most attacks can only be detected by examining a stream of sensor outputs. To capture this more sophisticated analysis technique one must use the following Goal spec:

```
Sw2 = spec
  sort P, Y, Q1, Q2
  op g: P -> Y
```

```
  op s1: P -> Q1
  op s2: P -> Q2
  op F: (P->Q1)*(P->Q2)->(P->Y)
  axiom FA is fa(p:P) g(p) = F(s1,s2)(p)
endspec
```

In this specification the sort (P->Q1) is the set of all functions from P to Q1. The op F maps a pair of functions to another function. This abstracts the notion of combining the results of two programs. The goal program g is computed by applying F to the two sensor programs. This is stated in axiom FA.

The colimit of the diagram in Fig. 1 is refined to a concrete spec in a series of refinement steps. The first step is to refine the sensor specs S_1 and S_2 using Sensor1 and Sensor2, as follows:

```
Sensor1 = spec
  sort Dump, Protocol
  op s1: Dump -> Protocol
endspec

Sensor2 = spec
  sort Dump, Address
  op s2: Dump -> Address
endspec
```

Note that the sensors extract information from the individual entries of the tcpdump. However, a single entry is not sufficient evidence for a smurf attack as noted earlier. A smurf attack is a property of the whole list of tcpdump entries rather than any particular entry. Furthermore, neither sensor can detect a smurf attack by itself.

Having refined the sensor specs, the specifier searches the sensor library for an appropriate template. In this step the specifier can use the Spec Matcher tool. The tool compares the signatures of the abstract specs Sensor1, Sensor2 and Goal to templates in the library. It checks whether all the necessary sorts and operations can be matched and then checks whether the matches map sorts and signatures consistently. This step should result in finding the template shown in Fig. 1.

Now the library is searched for concrete sensor specs that match Sensor1 and Sensor2. Again, the Spec Matcher tool can be used here. The result is the following sensor specs: AddressSensor and ProtocolSensor. Due to space limitations we show only a part of the ProtocolSensor spec.

```
ProtocolSensor = spec
  sort Protocol
  sort TraceLine = String
  sort Dump = List TraceLine
  sort ProtSeq = List Protocol
  op protTypeSeq: Dump -> ProtSeq
  def protTypeSeq(d) = map protocolType d
  op protocolType: TraceLine -> Protocol
  def protocolType(tr) = stringToProtocol
    (getProtocolString(tr))
endspec
```

In the next step the template is instantiated with the concrete sensor specs. This instantiation is represented by morphisms between abstract sensor specs and concrete sensor specs and the composition of these morphisms. The result is a refinement of the template. As an example, the following is the morphism from the abstract sensor Sensor2 to the ProtocolSensor:

```
Sensor2_ProtocolSensor =
  morphism Sensor2 -> ProtocolSensor
  {P +-> Dump, Protocol +-> ProtSeq,
   s2 +-> protTypeSeq}
```

Then the library is searched for specs that need to be imported into the refined template so that the abstract Goal can be refined to a concrete spec. Again, the Spec Matcher tool can be used here. In our example, the concrete specs that need to be imported and then translated are IPv4Addrs and Protocols. At this time the colimit operation can be invoked. This functionality is supported by Specware and thus does not require too much involvement by the specifier beyond simply requesting it. The outcome is a complete spec. In our case the spec was 287 lines long, so it is not possible to show this spec in this paper.

9 Code Generation

The final step is code generation. This operation, again, is supported by Specware. So all we need to do is to invoke the :swl command on the final spec. Specware generates Lisp code. For this example 1016 lines of code were generated.

We then were able to run this lisp code and see that it performed according to specifications. Here is an example of the invocation of the program and the result (we used an example that bases the detection decision only on two lines, but the program can accept any number of lines). In the first case there is no smurf attack; it is indicated by "NIL" given as output. In the second case, on the other hand, there is an attack, indicated by the "T".

```
  [1] SW-USER(51): (SmurfAttack? '(
"00.00:05.327 spoofed.target.com >
192.168.15.255: icmp: echo request"
"00.00:05.327 spoofed.target.com >
129.11.1.255: icmp: echo request"))
NIL

  [1] SW-USER(52): (SmurfAttack? '(
"00.00:05.327 spoofed.target.com >
192.168.15.255: icmp: echo request"
```

```
"00.00:05.327 spoofed.target.com >
192.11.1.255:  icmp:  echo request"))
T
```

10 Conclusions

The main point that we wanted to address in this paper is that of automating the most difficult part of information fusion – algorithm development – by the computer. We proposed to use a formal method approach in which a formal specification is developed first and then code is generated by a computer tool. In such a case, the automation focus is on the specification step. Towards this aim, we are developing a tool – Spec Matcher – that will help the specifier to perform semi-automatic specification development. In this paper we outlined all the necessary steps of such a process. For this purpose we used an intrusion detection example (detection of the smurf attack). The steps have been formalized in the MetaSlang language and verified, including automatic code generation, using the Specware formal specification tool.

Acknowledgments

This research was inspired by a grant from the Air Force Office of Scientific Research under contract No: F49620-98-1-0043 and by a grant from Air Force Research Laboratory, Rome, NY under contract No: AFRL-IF-RS-TR-1999-240. The authors wish to express their special thanks to Robert Paragi from AFRL for his help with the idea of using the information fusion approach to intrusion detection.

References

[1] A. N. Steinberg and C. L. Bowman. Revision to the JDL data fusion model. In D. L. Hall and J. Llinas, editors, *Handbook of Multisensor Data Fusion*, pages 2–1 – 2–19, CRC Press, 2001.

[2] T. Gruber. What is an ontology? 2006. www-ksl.stanford.edu/kst/what-is-an-ontology.html.

[3] Joaquin Miller and Jishnu Mukerji (Eds.). MDA Guide, Version 1.01. Technical Report omg2003-06-01, OMG, 2003.

[4] C. P. Pfleeger. *Security in Computing*. Prentice Hall, 2003.

[5] D. L. Hall and J. Llinas. An introduction to multisensor data fusion. *IEEE Transactions*, 85, No. 1:6–23, 1997.

[6] R. A. Kemmerer. Integrating formal methods into the development process. *IEEE Software*, 9:37–50, 1990.

[7] Formal methods specification and verification guidebook for software and computer systems. Technical Report NASA-GB-002-95, National Aeronautics and Space Administration, 1995.

[8] Martin D. Fraser, Kuldeep Kumar, and Vijay K. Vaishnavi. Strategies for incorporating formal specifications. *Communications of the ACM*, 37, No.10:74–85, October 1994.

[9] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 9:8–24, 1990.

[10] J. Rushby. Formal methods and the certification of critical systems. Technical Report CSL-93-7, SRI International, 1993.

[11] J.C.P. Woodcock. The rudiments of algorithm refinement. *The Computer Journal*, 35(5):441–450, 1992.

[12] B. C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[13] Specware 4.0 user manual. Technical report, Kestrel Institute, 2003.

[14] J. Goguen. Categorical foundations for general systems theory. *Advances in Cybernetics and System Research*, pages 121–130, 1973.

[15] J. A. Tomasik and J. Weyman. Category semantics for fusion and renement of multisorted specications. In *Proceedings of Fusion'2006, International Conference on Information Fusion*, 2006.

[16] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.

[17] S. A. DeLoach and M. M. Kokar. Category theory approach to fusion of wavelet-based features. In *Proceedings of the Second International Conference on Information Fusion, Vol. 1*, pages 117–124, 1999.

[18] M. M. Kokar, J. A. Tomasik, and J. Weyman. A formal approach to information fusion. In *Proceedings of the Second International Conference on Information Fusion, Vol. 1*, pages 133–140, 1999.

[19] M. M. Kokar, J. A. Tomasik, and J. Weyman. Data vs. decision fusion in the category theory framework. In *Proceedings of FUSION 2001 - 4th International Conference on Information Fusion, Vol. 1*, pages TuA3–15 – TuA3–20, 2001.

[20] M. M. Kokar, J. A. Tomasik, and J. Weyman. Formalizing classes of information fusion systems. *Information Fusion: An International Journal on Multi-Sensor, Multi-Source Information Fusion*, 5,3:189–202, 2004.