

The Opportunity for Formal Models of Integration

J. Payton R. Gamble S. Kimsen L. Davis

Dept. of Mathematical and Computer Sciences

University of Tulsa

600 S. College Avenue

Tulsa, OK 74104 USA

+1 918 631 2988

{payton, gamble, sonali, davis}@euler.mcs.utulsa.edu

ABSTRACT

Major industrial and governmental efforts in information sharing and integration require building or migrating applications using heterogeneous component systems. This style of software development enjoys the benefits of reusability, adaptability, and evolvability. However, as with most component integration attempts, interoperability problems arise. Although there are several strategies that currently exist for the integration of systems, many suffer from informality and are tightly coupled to particular domains and products. More importantly, interoperability problem detection and integration solution design show promise as areas where formal methods can be applied. In this paper, we discuss why the opportunity to use formal models of integration should not be overlooked. We present several criteria for formal modeling to be useful and usable for information and process integration. We illustrate our approach to integration modeling and discuss the expansion efforts needed to satisfy the criteria discussed.

Keywords: Component-based design, Integration, Interoperability, Formal methods

1 INTRODUCTION

As software ages and new paradigms for computing come to the forefront, the complex reuse, sharing and integration of information in terms of data, processes, and independent subsystems is becoming a paramount concern. The current industrial solution to integrate these types of heterogeneous software components is the use of off-the-shelf (OTS) middleware products [4] and Enterprise Application Integration (EAI) tools. These products are targeted toward integrating applications throughout a network or a distributed system, allowing clients access to information and processes transparently [4]. A product may implement one or more middleware frameworks to provide integration solutions, such as publish-and-subscribe and object request broker. Other types of middleware frameworks include remote procedure call (RPC), which provides transparent, distributed procedure calls; message-oriented middleware (MOM), which provides messaging capability between applications; and extract-transform-load (ETL) which moves, copies and/or merges data between

databases. Though this commercial effort is to be applauded, the idea that there is an OTS solution to all interoperability problems is misleading. There is a combination of middleware frameworks, tools, and techniques that can be applied, but their method of application is unique to specific problems.

Though there are instances where middleware products can be quickly deployed to resolve an interoperability problem, there are also many failures associated with their use. A major culprit is the limited understanding of the individual integration solutions that comprise the middleware products. Thus, the product becomes unwieldy to use, requiring a customized implementation by the tool vendor or an "expert" consultant. Though initial implementation costs may be reasonable with respect to the timeliness in which the integration can be performed, the lack of comprehension of the internal functionality can cause long-term difficulties. These difficulties emerge with system evolution, upgrades, and further need for integration, resulting in a significant increase in cost. Additionally, performance and reliability can suffer because it is difficult to show how a middleware product implementation satisfies requirements, such as timely access to information or correct information flow.

The key to understanding how to integrate components resides in the knowledge that middleware products and their underlying frameworks can be decomposed into small functional units responsible for resolving low-level interoperability problems (such as conflicting data representations) [11]. It is at this level that formal modeling and analysis can provide direct benefits to analyzing component qualities for interoperability conflicts and determining which functional unit resolves each conflict. In turn, knowing what integration functionality is actually needed enables the software engineer to make an informed decision about a middleware product's usefulness, i.e., which tools embody that required functionality. While we have defined some baseline requirements of these entities [11], descriptions are needed to show interoperability conflict resolution, property guarantees and composition into integration frameworks and their associated middleware products. However, to provide in-depth analysis, it is necessary to model these entities in a workable form that is at the same level of detail as an abstract component description, namely its software architecture.

Software architecture provides a suitable level of abstraction for formal modeling without "over formalizing," because of its coarse-grained descriptions of computational entities and their allowable connectivity within an application. Architectural styles and patterns have been modeled formally and informally with respect to the underlying components or modules, and connectors [1, 9, 11]. Furthermore, current research is defining architecture characteristics whose comparisons point to potential interoperability problems [10]. As patterns of conflicts emerge, defining mappings from conflicts to integration solutions is the next step. Thus, for seamless analysis and design of the integration solution, it is necessary to have

defined and modeled those functional units that form the integration solutions, all at the same architectural level of abstraction.

In this paper, we discuss the opportunities for formal models of software architecture descriptions for integration solutions, henceforth referred to as integration architectures, by modeling the basic functional units, their variances, and their composition. We present sample models using a combination of the Unified Modeling Language (UML) and Object-Z.

For the purpose of this paper, we use the following terminology (see Figure 1). A module is a component internal to an independent system that is participating in the integrated application. A component is the participating system. The application is the integrated system of components. A connector links modules within each component. An integration architecture is a special kind of “mega” connector that encapsulates the integration solution to link components in the application.

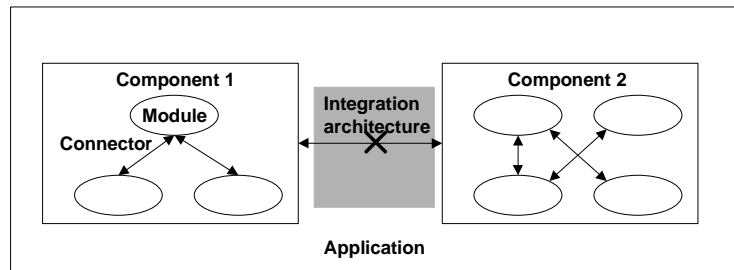


Figure 1: Integrated System Terminology

2 FOUNDATIONAL RESEARCH

The software architecture of a system describes its main computational entities (components), how they relate to each other (connectors) and constraints on their interaction (configuration) [17]. Data, control, and system characteristics are associated with software architecture. From this foundation, we have consolidated known component qualities that contribute directly to architecture interoperability [10], allowing for their comparison across multiple components to predict conflicts. Our objective is for integration concerns to be addressed early in the development of a component-based system leading to design solutions for integration frameworks.

Middleware (or integration) frameworks, such as CORBA, shared repository, and publish-and-subscribe, can be appropriately used to integrate heterogeneous software components [4, 5]. The use of frameworks can be problematic, however, when limited knowledge of their implementation hinders the ability of a developer to maintain the integrated application as it evolves. A better understanding of the fundamental entities that resolve low-level data- and control- related

interoperability conflicts can remedy this problem. In addition, the ability to understand how the composition of these entities performs (as embodied in an integration framework) will lead to better middleware product choices.

Integration frameworks are made up of underlying strategies and patterns [11, 12]. We use the term *integration architecture* to describe those strategies and patterns that make up a complete integration solution for a particular application. The integration architecture is different from the middleware product because it does not include extraneous functions, i.e., functionality that is not directly related to the known interoperability problems. Describing an integration architecture at the same level of abstraction as the software architecture of the component system requires expressing minimal functional units to resolve low-level conflicts. Different compositions of these building blocks (and their variations) form integration architectures [9], which in turn provides a comparative analysis of frameworks. We call these minimal functional units *integration elements* [11].

We have previously defined three integration elements: the *translator*, the *controller*, and the *extender*. The basic function of a translator is to convert data and functions between component formats and perform semantic conversions. A controller integration element coordinates and mediates the movement of information between components using predefined decision-making processes or strategies. The translator and controller elements are indivisible connector models of integration functionality. The extender is a connector model whose functionality can vary extensively, (e.g. buffering, polling, and security checks) and is specifically used as part of an integration architecture in which the translator and controller cannot accommodate the full functional need.

Interoperability problems often require more than one integration element as their solution. Integration architectures are thereby constructed by composing sets of integration elements [11]. To illustrate an integration architecture, assume we have a variety of components that all access data differently – some pass data directly, but most rely on an external source for data. The main interoperability problem is how to consolidate the data into one source that is both accessible to all components and can be altered for acceptance by all components. A shared repository framework can resolve this problem. The shared repository [15, 17] utilizes all three integration elements. For instance, an extender models the database (such as would be implemented in Oracle). Multiple translators, one for each component that requests access to the database, change the request to and from the components and database. Controllers determine the order in which requests of the components are given to the database, as well as deciding which translator (and hence, parent component) receives the information from the database after the component request has been processed.

3 FORMAL MODELING IN PERSPECTIVE

It is very likely that new processes will be developed that automatically compare the components in some manner, detect what potential conflicts are present, and recommend an integration solution strategy, as research and engineering are heading in that direction. But it remains to be seen (1) how the right level of granularity will be determined for the conflicts and their solutions, (2) whether there will be a direct relationship between the solution and middleware, and (3) what guarantees can be made about the solution. In an effort to thwart ad hoc approaches to the problem, we turn to formal methods, which, with some care, can be a viable approach to conflict detection and solution design. In this section, we focus on the issues surrounding the formal modeling of integration solutions; specifically addressing the qualities of a useful and usable modeling approach for acceptance in this area, and where current research is headed.

3.1 Criteria for usable and useful formal methods

While formal modeling of the primary entities present in middleware frameworks can offer in-depth insight into its internal makeup, we believe there are certain criteria that must be in place for integration modeling and analysis. They are as follows:

- a) A consistent and employable formal notation to describe the models of components, conflicts, integration elements, and integration architectures, such that they facilitate reuse through inheritance and refinement.
- b) An informal or semi-formal modeling approach used in conjunction with an underlying formal foundation to minimize complexity and increase transparency.
- c) A transformation method from the informal or semi-formal models to the formal models, in which properties can be expressed (possibly expanded) and proven at a formal level based on assertions at the informal or semi-formal level.
- d) Reusable templates for expressing similar architectural abstractions within the modeling languages to allow for consistency across entities, to facilitate inheritance and property guarantees, and to form a critical mass of models for a "plug-and-play" type approach, eliminating the need to model every element from scratch.

3.2 Where Formal Methods Stand

We can assess the maturity of formal methods research by examining the following important areas that contribute to the success of any component-based software design and analysis method, namely documentation, abstraction, flexibility and transparency, and specification of integration solution.

Documentation. Providing documentation of the interoperability conflicts that lead to specific middleware implementations

is essential for application maintenance. The availability of high-level, understandable formal models aids design documentation [3] by expressing the constraints and expectations that implementations must meet. As integrated systems evolve, through component upgrades or the insertion of new components, the model can be directly referenced to obtain past integration solution decisions and to determine if constraints have been violated. The key criteria is that the models must be understandable or in an automated form for referencing. Most models are advancing toward this point, but more research is needed. Thus, the use of combined informal or semi-formal notation is a must.

Abstraction. The advent of software architecture analysis provides a level of abstraction that can be formally expressed in a straightforward manner. Often formal methods are rejected because a designer/developer must always begin the modeling from scratch. Architecture description languages (ADLs) have been developed to express the abstract nature of software architecture designs. Each ADL tailors its specification capability toward specific types of analysis. For example, two predominant ADLs that utilize formal specification and proof are Wright [2] and Darwin [13]. The use of either of these ADLs as they mature will likely result in more accurate specialized analysis. Additional ADLs are needed for a complete interoperability analysis.

Flexibility and transparency. Although it is recommended to choose an appropriate notation [3], it is rare that a single language incorporates all the features desired for the model. Several languages may be needed depending on the analysis required. Different notations allow for a broader description and analysis of integration properties and guarantees. In other words, it is possible to describe components of a software system using a graphical modeling language and have a workable representation in a formal model [6]. Properties can be added to the formal model to express constraints and requirements that can be proven. As interchange languages [8] emerge, they will represent multiple notations to facilitate specific property expression and proof, depending on the type of analysis required.

The UML [16] has emerged as a standard notation for modeling software systems. The UML provides a visual language for specifying a system as a composition of many model elements, with its foundation in object-oriented design. Using the UML to model middleware solutions employs class diagrams that aid developer understanding of the configuration and coordination of the participating components. Relationships among model elements give insight to the associations and dependencies among classes. For the purpose of specifying architecture components and integration solutions, the UML can serve as a semi-formal modeling language [14, 18]. The UML supports inheritance and binding, but, unfortunately, lacks reasoning and verification [6].

The UML has an associated Object Constraint Language (OCL) [16] for formal modeling of certain properties. While OCL can be used as a navigational language, it is not as expressive as ADLs or other formal languages. Thus, research

has been conducted to translate models from the UML to Object-Z [6], which can capture the constraints on syntactic structures that can be formed using UML diagrams. It is possible to use the axiomatic definitions in Object-Z to express fairness properties of a component. In addition, the logic for Object-Z can be used to express the safety and liveness properties of a complex specification using the temporal logic predicates. Within the strict modular semantics of Object-Z, notions of object properties and system properties can be easily formalized and understood. Object-Z supports inheritance and refinement, and overcomes many of the problems associated with OCL. Thus, a combination of the UML and Object-Z has the potential to serve as a flexible, dual representation method.

Specification of integration solutions. Previous research shows the ability to model elements of integration as embedded within an architectural style [7]. Our definition of the integration elements [11] provides the impetus for developing formal models because of the minimal functionality each element has. Restrictive theories of composition have been developed for components in the ADLs Wright [2] and Darwin [13]. Research is ongoing to determine if these theories can be extended to Object-Z models so that integration elements can be provably composed into integration architectures that satisfy the set of interoperability problems detected. As we show in the next section, these elements can be formally modeled and combined in a manner that allows for guarantees on the integration architecture defined.

4 A POTENTIAL APPROACH

Referring to the criteria identified for a set of useable and useful formal models, it is apparent that formal methods research is in position to bring value to integration modeling and analysis. In this section, we describe an approach to modeling integration frameworks based on the view of formal modeling and the research discussed in the previous section. The approach relies on templates to model generic integration elements. Refinement and inheritance on the basic templates are used to create a variety of special purpose elements. The use of the UML to maintain the architecture descriptions of the templates provides a graphical display of their organization. Figure 2 shows the component hierarchy tree using the generalization and dependency relationships.

4.1 Representing the Integration Elements in the UML

An abstract `Architectural_Component` forms the root of the tree in Figure 2. The `Controller` and `Extender` are also abstract types because of the need for different underlying templates¹. The `Controller`'s abstract class is partitioned into three distinct types that vary according to how their decisions are made. For instance, the `Composer_Controller` bases its decisions on which input components provide information and on the type of information received. The `Distributor_Controller` makes

¹ It is possible that the hierarchy may change with respect to the translator as our understanding of its variations matures.

decisions on which output receives information and what information is passed. The Coordinator_Controller is a bi-directional controller that can both composes and distributes. Because the Extender has many different functions, it currently is represented as an abstract class. The templates (Translator, Composer_Controller, Distributor_Controller, Coordinator_Controller) have variable types that are bound according to the application requirements. Intermediate classes represent type bindings. The leaf nodes of the tree represent specialized integration elements for the shared repository architecture as discussed in Section 2.

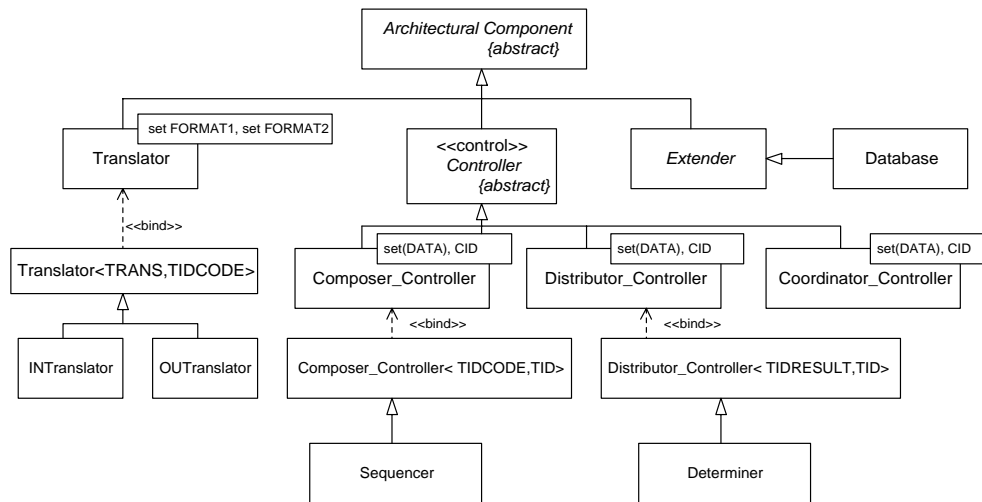


Figure 2: The UML Template and Class Diagram

Assume that multiple interacting components conflict with respect to data exchange. One aspect of the conflict may be that they have different data representations. Another aspect may be that the components use distinct repositories to obtain data that then overlaps as components are combined into a single application. Thus, a potential integration solution would include a shared repository architecture. Uniform modeling of the shared repository provides an understanding of the underlying building block functions and how they work together to form a complete solution. In addition, it provides a basis from which to construct the formal model. The UML for this integration architecture is shown in Figure 3.

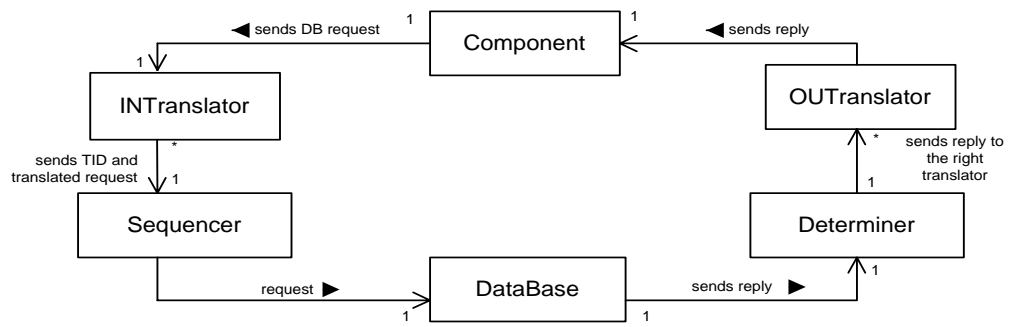


Figure 3: Shared Repository Diagram using Integration Elements

Composition of integration elements presents a unique challenge. Work is continuing to formulate a theory of composition of integration elements and what this means to the resulting integration architecture. For the purpose of illustrating the formal modeling approach, we focus more on defining the integration architecture for this example. Therefore, we have separated what would normally be a bi-directional translator to and from each component and the database into two uni-directional translators that embody a single mathematical function.

As shown in the class diagram of Figure 3, every component is associated with one INTranslator and OUTTranslator, which are both types of Translators (refer to Figure 2). The Sequencer, which is a type of Composer_Controller (see Figure 2), is fed input by multiple translators and determines the sequence of passing requests to the database. The database is an instance of an Extender that represents the merging of multiple databases. A Determiner, which is a type of Distributor_Controller, passes the results to translators (via OUTTranslator types) linked to the components.

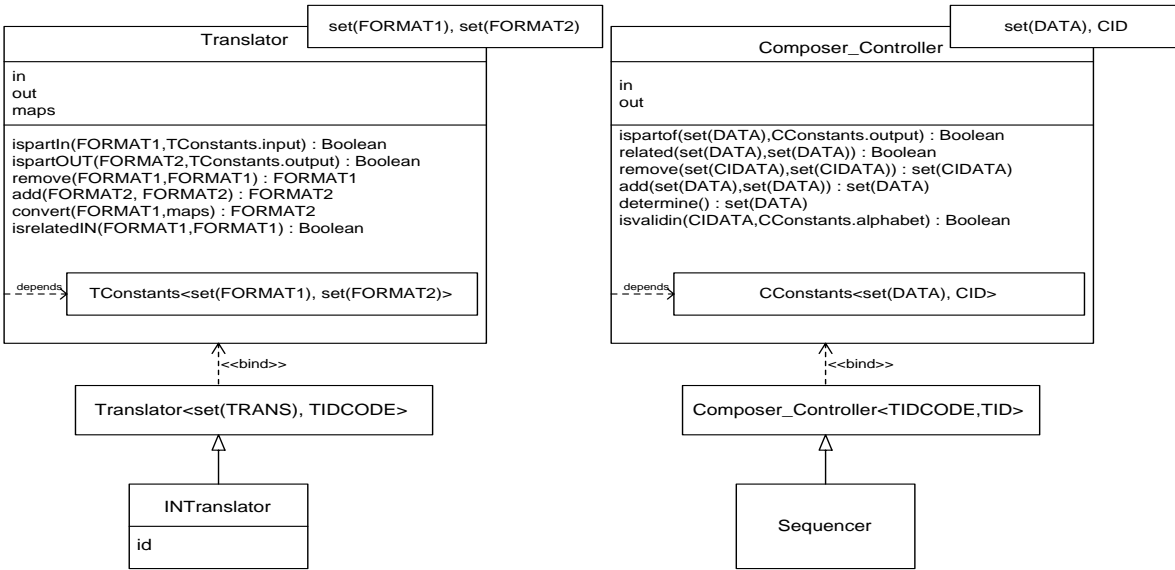


Figure 4: Composer Controller and Translator Templates using the UML

4.2 The Formal Modeling of an Integration Solution

Figure 4 shows the Composer_Controller and Translator classes in more detail. A side-by-side display illustrates the similarities of the object representation based on the templates. The Object-Z models that appear in Figures 5 and 6 directly reflect the input/output ports, collection variables, constants, and type bindings seen in the UML models. Furthermore, the formal models provide constraints on the variables that cannot be stated in UML. Inheritance indicated by the open arrows in Figure 4 is shown in the first line inside the schema box in Figures 5 and 6.

The parts of the Object-Z schemas are labeled to the left of the schema box. On the right, in italics, are comments regarding the formal expressions. For brevity, we omit the detailed definitions of the generic functions shown in bold and the constants associated with each class. Specific to Object-Z models is that the variables x and x' represent the before (no prime) and after (prime) states, respectively, of the variable x . This allows for accurate logical expressions along with conveying the meaning of assignment. Additionally, the notation Δx , means that only the state variable x will change as a result of an operation (Step as in Figures 5 and 6). While the models may seem cumbersome, the goal is to express a critical mass of models that are only manipulated to accommodate variations rather than completely writing a model each time. As discussed in Section 3, translation from UML to Object-Z is currently manual. Unfortunately, complex type definitions are cumbersome if not impossible in UML. Hence, even with an automated translation, currently these type definitions would need to be manually inserted in Object-Z models.

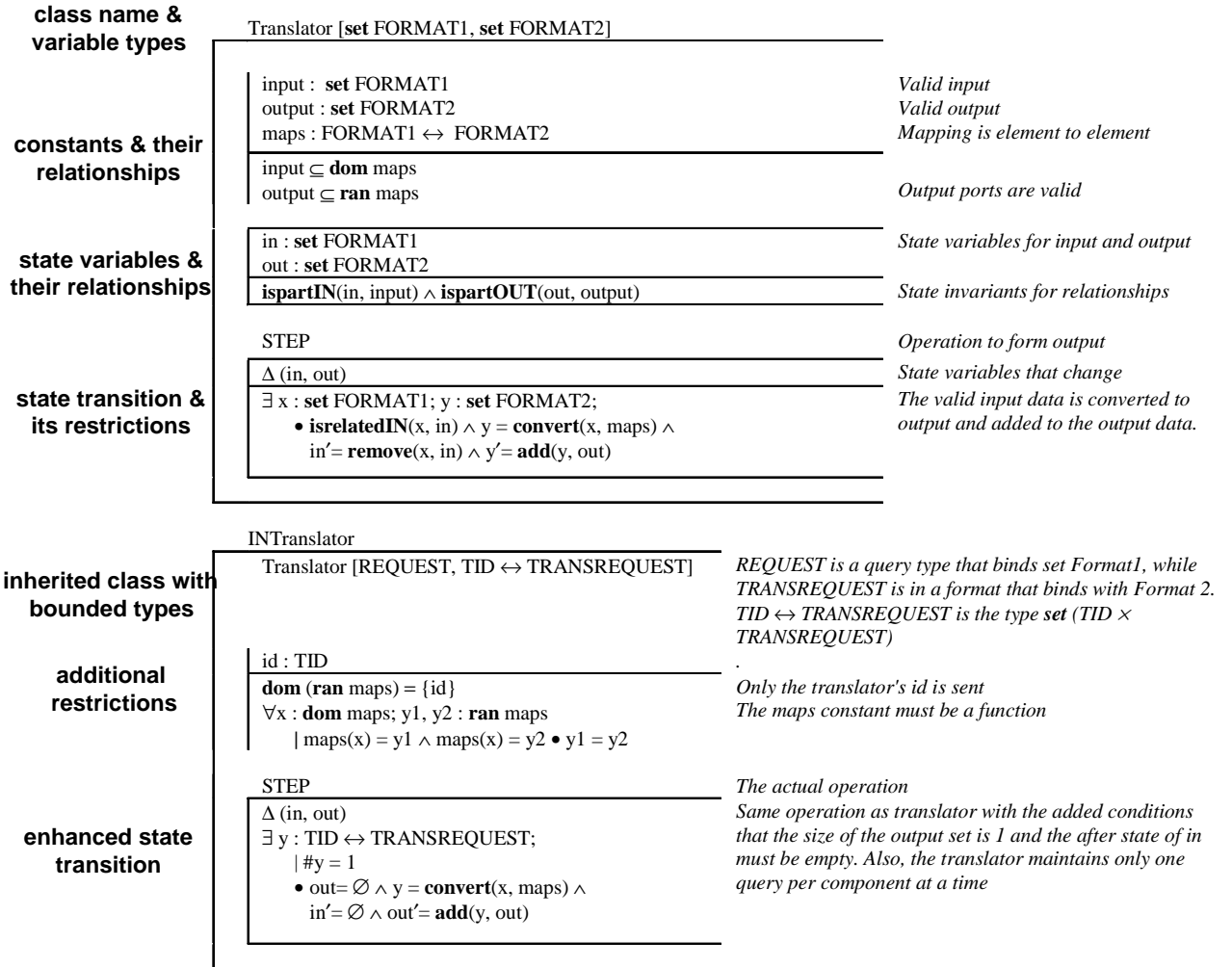


Figure 5: Formal Model of Translator and Specialized INTranslator

It is important to note that both model types have a set of constants that detail the allowable input and output data (namely, *input* and *output* for the Translator and *inports* and *outdata* for the Controller) and how they are modeled (see the inserted boxes in Figure 4 and the first open schema in Figures 5 and 6). For instance, in the Composer_Controller, the input ports are explicitly named (through *inports*). There is a set of acceptable names for these ports as well as allowable data for each one represented by constraints on the constant *alphabet*. For the Translator, all incoming data is collected into a single variable (*input*) from the ports because the Translator does not need to know who sent the data to perform its transformation. Both templates have variables (*in* and *out*) representing the current state of the input and output, respectively. Both have a set of abstract functions that are used to relate *in* to *out*, as well as relate *in* and *out* to the constants *input* and *output*, respectively. Because data can be in many formats and input/output can be processed multiple ways, these abstract functions

provide flexibility.

In Figures 4, 5, and 6, the definitions become more concrete as variations and specializations are modeled, though this can be seen best in the Object-Z models. Refinement of the INTranslator and Sequencer models reflects the application requirements enforced by the shared repository. For example, the Translator constant *maps* is a relation. However, INTranslator in Figure 6 and OUTTranslator restrict this relation to a function. As defined in Section 2, the Controller does not perform any translation. Thus, its input and output data types are the same. This requirement is maintained in its refinement to the Composer_Controller and Distributor_Controller and downward through the hierarchy.

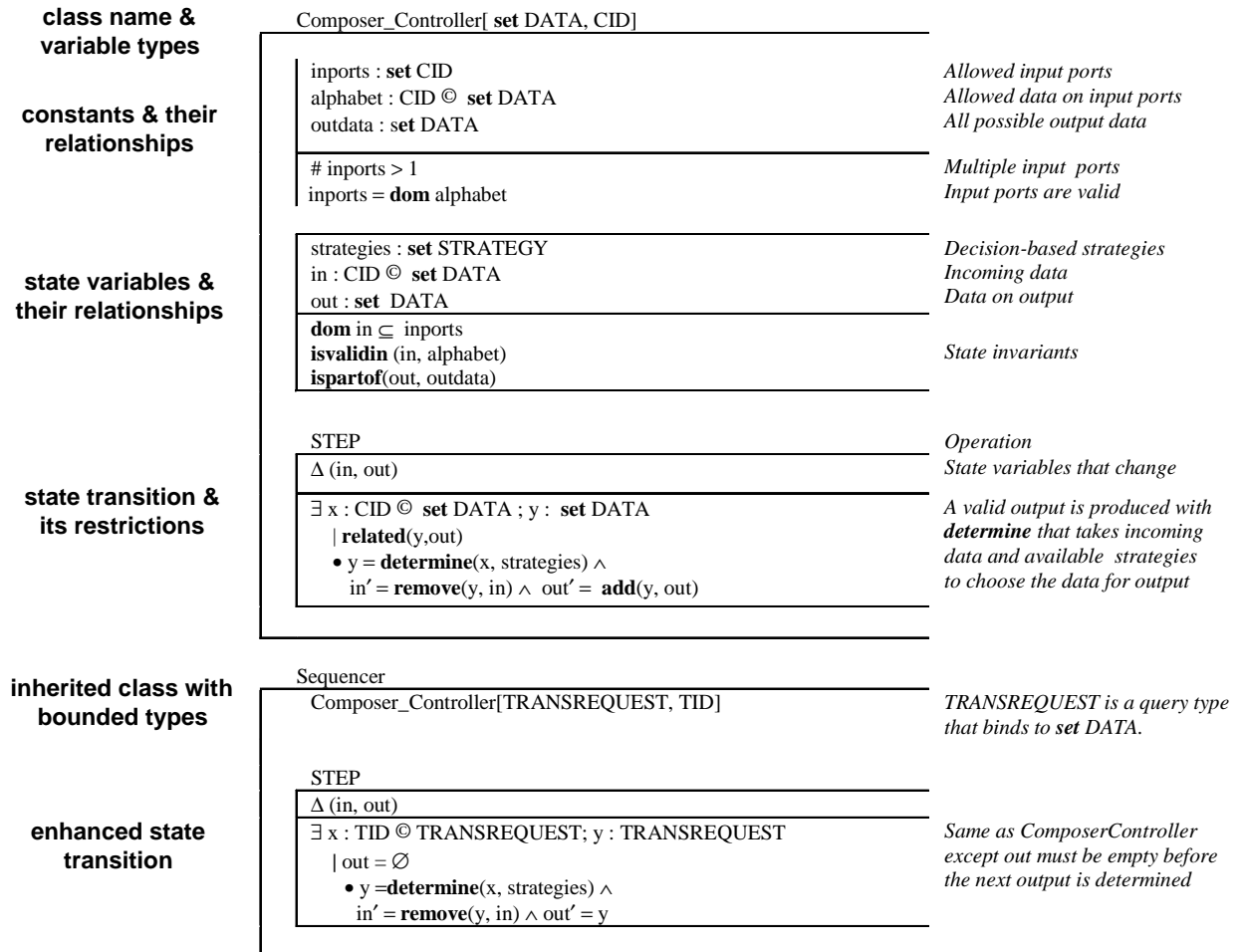


Figure 6: Formal Model of Composer Controller and Specialized Sequencer

The templates for these models are loosely based on earlier models of the pipe and filter and event-based styles [1]. In these models, connectors were explicitly specified to transfer the data between components. We have since normalized the templates to take advantage of inheritance and refinement. The models presented in Figures 5 and 6 do not require explicit

connectors. Rather, transfer operation schemas act as connectors to describe the passing of the output data from one class to the input of another. We illustrate the transfer operations below. Note that in each case, a set of discrete data is being passed.

From a component to INTranslator where COM is of type Component and T1 is of type INTranslator.

TransferCOMtoINT

Δ (COM.out, T1.in)
$\exists r : \text{REQUEST}$ $ r = \text{head COM.out.dbrequest} \wedge T1.in = \emptyset$ <ul style="list-style-type: none"> • COM.out.dbrequest' = last COM.out.dbrequest \wedge T1.in' = COM.out.dbrequest

Variables from each entity that change.

The request, (the first in the output sequence of the component) is accepted as input by the translator provided that all previous data has been processed.

The request is deleted from the component's queue and placed as input to the translator.

From INTranslator to Sequencer, where T1 is of type INTranslator and SEQ is of type Sequencer.

TransferINTtoSEQ

Δ (T1.out, SEQ.in)
$\exists t : \text{TID}; q : \text{TRANSREQUEST}$ $ \{(t, q)\} = T1.out \wedge t \notin \text{dom SEQ.in}$ <ul style="list-style-type: none"> • T1.out' = $\emptyset \wedge$ SEQ.in' = SEQ.in $\cup \{(t, q)\}$

Variables from each entity that change.

INTranslator passes entire transformed query to Sequencer. Sequencer will block input if data from that translator has not yet been processed. Successful transfer deletes the query from the translator and adds it to Sequencer input.

From Sequencer to the Database, where SEQ is of type Sequencer and DB is of type DB.

TransferSEQtoDB

Δ (SEQ.out, DB.in)
$\exists t : \text{TID}; q : \text{TRANSREQUEST}$ $ \{(t, q)\} = \text{SEQ.out} \wedge$ <ul style="list-style-type: none"> • SEQ.out' = SEQ.out $\setminus \{(t, q)\} \wedge$ DB.in' = DB.in $\wedge \langle (t, q) \rangle$

Variables from each entity that change.

The database requires its input to be queued as a sequence. The Sequencer removes the passed information from its output and appends it to the database input queries using sequence concatenations.

From the Database to Determiner, where DB is of type Database and DET is of type Determiner.

TransferDBtoDET

Δ (DB.out, DET.in)
$\exists t : \text{TID}; p : \text{REPLY}$ $ (t, p) = \text{head}(DB.out) \wedge t \notin \text{dom DET.in}$ <ul style="list-style-type: none"> • DB.out' = last(DB.out) \wedge DET.in' = DET.in $\cup \{(t, p)\}$

Variables from each entity that change.

Determiner collects responses from DB, but only one per translator id. This may cause the DB to block as it waits to pass its reply.

From Determiner to OUTTranslator, where DET is of type Determiner and OUT is of type translator.

TransferDETtoOUT

Δ (DET.out, T2.in)
$\exists t : \text{TID}; p : \text{REPLY}$ $ t = T2.id \wedge (t, p) \in \text{DET.out} \wedge T2.in = \emptyset$ <ul style="list-style-type: none"> • DET.out' = DET.out $\setminus \{(t, p)\} \wedge T2.in' = \{(t, p)\}$

Variables from each entity that change.

The output of the Determiner is sent to the correct OUTTranslator according to its ID. The OUTTranslator input must be empty to receive a response.

From OUTTranslator to component,

TransferOUTtoCOM

Δ (T2.out, COM.in.dbresponse)
$\exists y : \text{TRANSREPLY}$ $ y = T2.out \wedge$ <ul style="list-style-type: none"> • T2.out' = $\emptyset \wedge$ COM.in.dbresponse' = COM.in.dbresponse $\wedge \langle y \rangle$

Variables from each entity that change.

The OUTTranslator sends inserts its translated response into the queue of its parent component through sequence concatenation.

4.3 Guaranteeing Integration Solution Requirements

Assume that the shared repository integration architecture in Figure 3 must meet the following requirement.

All requests by a component receive a response.

If we assume that the INTranslator and OUTTranslator abide by the functionality specified, as well as the transfer

operations to and from the components to the translators, then we need only show the requirement is satisfied for one component to be satisfied by all components. It is easier to show that the individual elements satisfy the requirements by examining the uni-directional trace across the components. Additional assumptions that cannot be seen by the partial specification are:

- (1) The database maintains the same sequence of input requests in its output responses.
- (2) The database always provides a response, even if it is an error message.

The component and the database both maintain I/O ports that queue the queries and replies. There is exactly one INTranslator and one OUTTranslator associated with each component (even if translation is not needed.) Each translator must perform a complete translation before accepting more data. Thus, the Sequencer and Determiner can only accept data if previous data with the same TID has been processed. This may cause the database to wait on input from the Sequencer or to block on output to other Determiner. However, each transfer operation obeys the requirements and passes an entire request/reply when delivering data. By composing the operations across all links, a functional mapping can be defined from request to response, upholding the requirement.

5 CONCLUSIONS

In this paper, we examine the prospects of using formal methods to describe and analyze integration frameworks. We describe integration element solutions to interoperability, using both UML and Object-Z to provide a visual and formal representation, with the goal of satisfying the criteria defined in Section 3 for useable and useful formal models. Furthermore, we describe a shared repository integration architecture and demonstrate the use of the integration element templates and transfer operations within this architecture.

The work in this paper can have a positive impact in the resolution of data and process interoperability conflicts. Expansions of the models presented in the languages we use or other languages would be extremely beneficial in understanding and analyzing middleware. In addition, a uniform representation or interchangeable representations facilitate comparative analysis of integration solutions. Using formal models for integration allows for a documented history of constraints and expectations that the middleware implementation must meet.

Further research is still needed to address shortcomings with formal modeling. Translation from the UML integration element models to their Object-Z counterparts is needed so that the UML can be manipulated with the Object-Z changing in the background. Interchange languages must mature so that models described in both the UML and Object-Z can be transformed to other representations. Additionally, better formal expression of the configuration of the integration

elements into an architecture is needed to form more complete proofs.

Acknowledgement: This research is sponsored in part by AFOSR, contract #F49620-98-1-0217. The government has certain rights to this publication.

6 REFERENCES

1. G. Abowd, R. Allen, D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM TOSEM*, 1995.
2. R. Allen. A formal approach to software architecture, Ph.D. Diss., TR CMU-CS-97-144, 1997.
3. J. Bowen & M. Hinchey. Ten Commandments of Formal Methods, *IEEE Computer*, 28(4):56-63, April 1995.
4. J. Charles. Middleware moves to the forefront, *IEEE Computer*. 17-19, May 1999.
5. R. DeLine. Techniques to resolve packaging mismatch, *ICSE 99*, 1999.
6. A. Evans, R. France, K. Lano K, B. Rumpe. The UML as a formal modeling notation, *UML 98*, 336 – 348, 1998.
7. R. Gamble, P. Stiger, R. Plant. Rule-based systems formalized within a software architectural style, *Journal of Knowledge based System*, Vol. 12, pp. 13-26, 1999.
8. D. Garlan, R. Monroe, D. Wile, ACME: An Architectural description language, *Proceedings of CASCON '97*, 1997
9. K. Hasler, R. Gamble, K. Frasier, P. Stiger. Exploiting inheritance in modeling architectural abstractions, Position Paper, *1st Working IFIP, Conf. on Software Architecture*, March '98
10. A. Kelkar and R. Gamble, Understanding the architectural characteristics behind middleware choices, *Proc. 1st Conf. on Information Reuse and Integration*, Sept. 1999.
11. R. Keshav, R. Gamble. Towards a taxonomy of architecture integration strategies, *Proceedings of ISAW3*, 1998.
12. G. Larsen. Using patterns in the UML, *Communications of the ACM*, Vol. 42, No. 10, October 1999.
13. J. Magee, N. Dulay, S. Eisenbach, J. Kramer. Specifying distributed software architectures, *In Proceedings of the 5th European Software Engineering Conference*, Barcelona, 1995.
14. Nenad Medvidovic and David S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 161-182, San Antonio, TX, February 22-24, 1999.
15. D. Mularz. Pattern-based integration architectures. *PloP*, 1994.
16. Rational Software. <http://www.rational.com>.
17. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
18. V. Gruhn and U. Wellen, Integration of Heterogeneous Software Architectures – An Experience Report, *First Working IFIP Conference on Software Architecture*, pp. 22-24, February 1999.