Naval Research Laboratory

Washington, DC 20375-5320

NRL/MR/5542--18-9795

Decision-based Model Driven Software Development

James Kirby Jr.

Center for High Assurance Computer Systems
Information Technology Division

June 4, 2018

DISTRIBUTION STATEMENT A: Approved for public release. Distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)*
04-06-2018

2. REPORT TYPE
Memorandum Report

3. DATES COVERED *(From - To)*
July 2016 - November 2017

4. TITLE AND SUBTITLE
Decision-based Model Driven Software Development

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
James Kirby Jr.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
55-6010

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

8. PERFORMING ORGANIZATION REPORT NUMBER
NRL/MR/5542--18-9795

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Naval Research Laboratory
4555 Overlook Avenue, SW
Washington, DC 20375-5320

10. SPONSOR / MONITOR'S ACRONYM(S)
NRL

11. SPONSOR / MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A: Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This paper assumes a software-intensive system embedded in an environment of entities and their attributes. Behavior of the system, for which software is responsible, is its effect on selected attributes in the environment. A model useful to software development records relevant entities and attributes; distinguishes attributes the system affects; records that effect with sufficient precision to be executed; organizes it to maintain intellectual control and constrain the impact of change during development, sustainment, and assurance; and organizes it to make efficient use of processing and communication resources.

15. SUBJECT TERMS
Modeling, Decisions, Behavior, Ontology

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					James Kirby Jr.
a. REPORT	b. ABSTRACT	c. THIS PAGE	SAR	20	19b. TELEPHONE NUMBER *(include area code)*
Unclassified Unlimited	Unclassified Unlimited	Unclassified Unlimited			202-767-3107

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

This page intentionally left blank.

Decision-based Model Driven Software Development

James Kirby Jr.
US Naval Research Laboratory, USA

ABSTRACT

This paper assumes a software-intensive system embedded in an environment of entities and their attributes. Behavior of the system, for which software is responsible, is its effect on selected attributes in the environment. A model useful to software development records relevant entities and attributes; distinguishes attributes the system affects; records that effect with sufficient precision to be executed; organizes it to maintain intellectual control and constrain the impact of change during development, sustainment, and assurance; and organizes it to make efficient use of processing and communication resources.

Keywords: Modeling, Decisions, Behavior, Ontology

INTRODUCTION

This paper assumes that software development is a decision-making process. Source code reflects many of those decisions, but often doesn't capture them explicitly. Decisions may be captured explicitly in models, specifications, and other documents, including notes, email, and instant messages, but they may not be updated as code evolves during development and sustainment.

This paper considers decisions that developers of software-intensive systems make. It assumes that for such systems, software determines and controls system behavior. While the discussion of decisions proceeds in a linear manner, there is no implication that the decision-making process itself proceeds in this order. Rather, the paper assumes that software development and sustainment processes are iterative and opportunistic: decisions are repeatedly made, unmade, fixed, and replaced. The resulting representation and organization of decisions in a model "fakes" a rational process, as described by [13]. This discussion does not address how developers should make these decisions. Rather, it identifies decisions that developers should explicitly consider and provides guidance on how to record and organize those decisions.

Decisions that models capture

This paper assumes that it would be useful to base software models on decisions that developers of software-intensive systems make, which include:

1. **Environment.** What is the relevant environment of the system?
2. **Behavior.** What is the behavior of the system visible on its boundary with the environment?

Manuscript approved April 26, 2018.

3. **Modularization.** How to decompose the work of making behavior decisions to facilitate intellectual control, management of change, and rational decision making?
4. **System.** How to decompose the behavior to make efficient use of system resources, e.g., processors, communication links?

Guided by stakeholders, developers decide what is the relevant environment of the system, which may be organizational, physical, cyber, or a combination of two or more. To distinguish the system in its environment and record the boundary of the system with that environment, developers decide what are relevant entities (things) in the environment of the system, and what are relevant attributes (properties or characteristics) of the entities, which may include the system itself in whole or in part. The attributes establish the boundary of the system with its environment. The entities establish and communicate system context.

Developers decide what behavior of the system is visible at its boundary with the environment, which may include all or part of the system when, e.g., it must respond to its own failure. Recording the behavior of the system entails deciding which attribute values the system controls or influences, called *controlled attributes*. The values that the controlled attributes assume over time in response to a changing environment constitute the behavior of the system. A model of that behavior includes (mathematical) functions that control or constrain values the controlled attributes assume over time.

There are two alternative decompositions of behavior decisions, made using different criteria and serving different purposes. Developers decide how to decompose the work of making behavior decisions into pieces, called *modules*, to facilitate intellectual control and to constrain the impact of change, which is inevitable throughout development and operation. Developers decide what is the interface that each module presents to others.

Developers decide also how to decompose behavior into pieces, called *software components*, which can be assigned to processors to make effective use of processor, communication, and other resources. Developers decide what is the interface that one software component presents to others. Communication resources facilitate interfacing software components executing on different processors.

MODELING DEVELOPERS' DECISIONS

To describe developers' decisions, the paper develops ontologies, "a formal explicit description of concepts in a domain of discourse" [10]. The concepts here are developer decisions. Overlapping ontologies for each of the four categories of decisions introduced in the previous section describe four overlapping models. Each will be discussed in turn in the present section. *Appendix I* illustrates the combined ontologies.

Figure 1 illustrates an ontology for modeling environment decisions, adapting UML class notation. The boxes (called *classes*) represent decisions developers make. Open-headed arrows point from concepts to more general concepts. The former, called *subclasses*, represent a specialization of the latter. In Figure 1, the class *Environment*, which represents decisions about things in the environment of a system, is a specialization of *Decision*, representing developers' decisions. *Slots* listed in a class represent properties of

the class they label, contributing to capturing developer decisions. Slots are *inherited* by the class's subclasses. Properties may be named and typed. Examples are two slots for the *Environment* class, *name_* and *description*, which are strings giving, respectively, the name (the underscore is an artifact of the ontology tool used) and description of a thing in the relevant environment.

Properties may also be one or more instances of another class, redundantly indicated by a slot of type *relation* and a labeled dotted arrow to the other class. The second slot in *Environment*, *listOfAttributes*, is a relation. The dotted arrow labeled *listOfAttributes* from *Environment* to *Attribute* indicates that relation between instances of the two classes.

Appendix II and *Appendix III*, respectively, define the classes and slots.

Environment decisions model

Figure 1 illustrates the ontology of the Environment Model. The most general concept is *Decision* representing decisions about the system that developers of software-intensive systems make. This model and three other models described below specialize that concept. Specializations of *Decision* in this model are *Environment* and *Variable*, the former representing decisions about things or entities in the environment, the latter representing decisions captured by named, typed variables. While not indicated in the figure, all three are abstract classes that do not have instances, only subclasses. Instances of *Attribute* class, a subclass of *Variable*, represent

> environmental quantities to be measured or controlled and the representation of those quantities by mathematical variables. The environmental quantities include: physical properties (such as temperatures and pressures), the readings on user-visible displays, administrative information (such as the number of people assigned to a given task), and even the wishes of a human user. These quantities must be denoted by mathematical variables and, as is usual in engineering, that association must be carefully defined, coordinate systems, signs, etc., must be unambiguously stated. [14]

The *children* relation of *Environment* indicates that a thing in the environment may be comprised of other things. Subclasses of *Environment* indicate that things in the environment may be either the system (or a part of the system) being developed (*System*) or a thing that is neither the system nor a part of it (*NotSystem*). The system and things that are not the system can each be comprised of other things (indicated by *children* relations). In the case of an Earth satellite system (*System*), its ground station and the Earth might comprise parts of *NotSystem.* In the case of a health care system (*System*), a hospital, its staff, and patients might comprise parts of *NotSystem.* Further, decisions about the system can identify system components that are either software or hardware.

The string *interpretation* slot of *Attribute* captures the association of an environmental attribute and the associated variable described by the remaining slots (e.g., *type*, *name_*, *valueOfAttribute*). The *listOfAttributes* relation indicates that developers decide which attributes (environmental quantities) apply to which things.

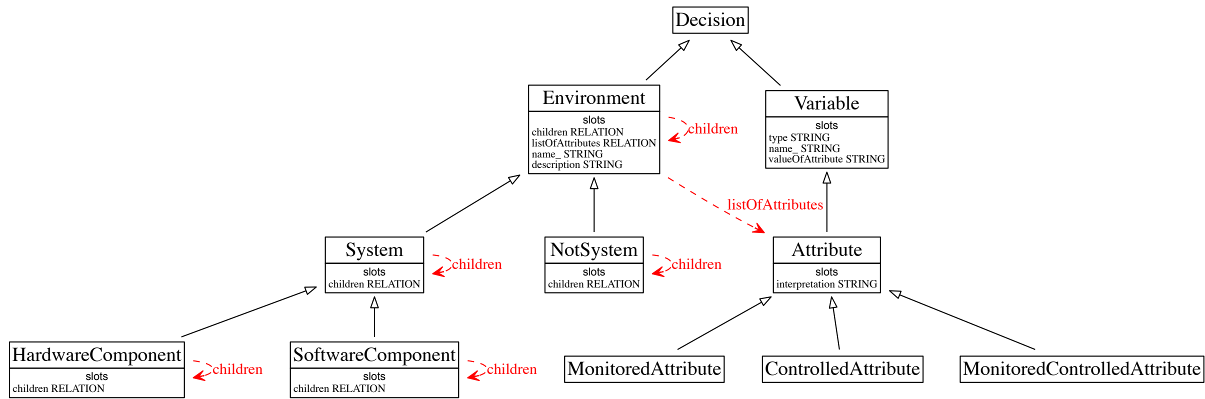

Figure 1 Environment Decisions

Developers may decide that an instance of *Attribute* may be a *MonitoredAttribute*, whose value the system can measure or sense, a *ControlledAttribute*, whose value the system can set or affect, or a *MonitoredControlledAttribute*, whose value the system can both sense and effect. The latter will not be further discussed.

Behavior decisions model

Figure 2 illustrates the ontology of the Behavior Model. Behavior of a software-intensive system embedded in an environment of things is understood to be the values that instances of C*ontrolledAttribute*, a specialization of *Attribute*, assume over time. The values of controlled attributes are functions of the values that other variables assume over time. Focusing on describing required software behavior in terms of output values sent by a computer, Heninger [7] describes

> software [behavior] as a set of functions associated with output [variables]: each function determines the values for one or more output [variables] and each output [variable] is given values by exactly one function. Thus every function can be described in terms of externally visible effects.

Heitmeyer and colleagues [6] formalize Heninger's semi-formal approach, representing each behavior function by a conditional assignment statement comprised of Boolean expression guards that select one value expression to compute the variable value. Reference [9] extends the approach to design specifications facilitating using one specification of behavior to serve both requirements and design.

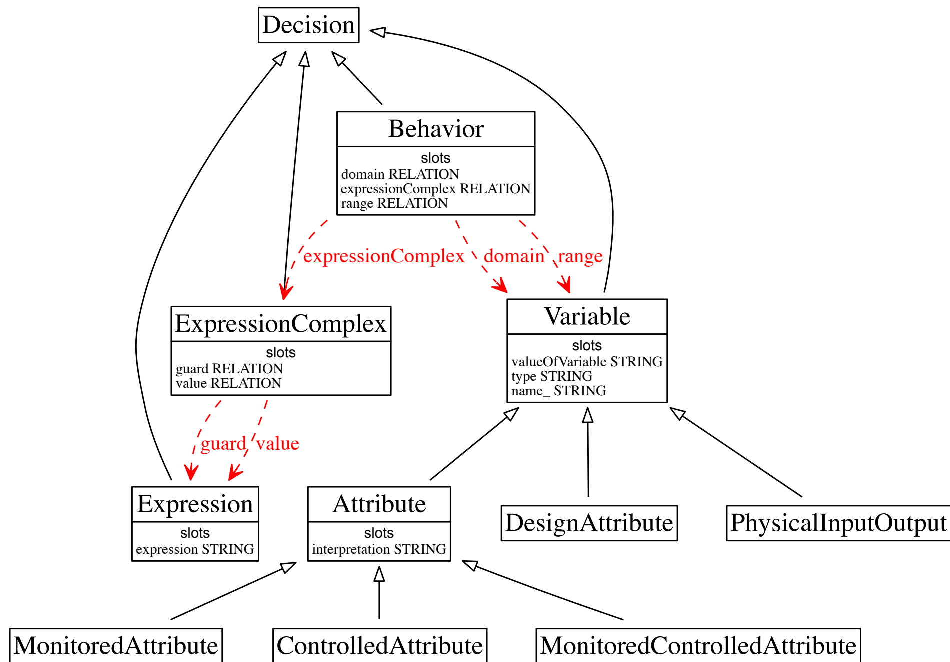

Figure 2 Behavior Decisions

Figure 2 introduces *Behavior*, *ExpressionComplex*, *Expression*, *DesignAttribute*, and *PhysicalInputOutput* classes. The figure illustrates how developers can capture software-intensive system behavior. Each instance of *Behavior* describes a function that calculates the value of the variable at the head of the arrow representing the *range* relation. Variables (multiplicity is not indicated by the figure) at the head of the *domain* relation comprise the domain of that function. The relation *expressionComplex* identifies the instance of the class *ExpressionComplex* related to an instance of *Behavior*. The relations of *ExpressionComplex* allow developers to record the guard and value expressions (written in terms of variables identified by the domain relation) of Heitmeyer's conditional assignment statement that compute the value of the variable in the *range* relation of *Behavior*. While the behavior decision model shares the *Variable* concept with the environment decision model, Figure 2 includes two subclasses of *Variable* that don't appear in Figure 1. *DesignAttribute* allows developers to identify additional variables that aren't attributes. For systems embedded in a physical or cyber environment, *PhysicalInputOutput* facilitates communicating with cyber or physical sensors and actuators that allow software to measure and control the environment. It will not be discussed further.

Modularization decisions model

Figure 3, which illustrates the ontology of the Modularization Model, introduces *Module* and *Interface* classes. Top level decisions in this model are recorded by *Module*, *Interface*, and *Variable* classes, the latter of which is shared with other decision models. The Modularization Model incorporates two important software structures: the *module* structure and the *uses* structure. [12] *Module* class and its *submodules* relation record the module structure, decomposing the work of software development and sustainment into a tree-structured set of work assignments called *modules*. A string *rationale* slot captures the rationale for each module. Reference [4] illustrates a fully worked out module

structure for the operational flight program (OFP) of a Naval aircraft. In the module structure of [4], the rationale is referred to as the module's *secret*, which describes decisions hidden or encapsulated by the module.

Decomposing complex systems into pieces can facilitate intellectual control. Depending upon the decomposition criterion, such decomposition can also facilitate ease of change [11]. In the Modularization Model, decomposing the work of making decisions involves developers implicitly assigning a behavior function to the module structure by associating the variable whose value it calculates with a module via the module's *local* relation or via the *output* relation of an *Interface* class associated with the module via the *requires* or *provides* relation. The *local* relation identifies variables that are local to a particular module. Their existence is known only to that module. The *provides* and *requires* relations identify variables that are known to multiple modules.

Parnas [12] defines *uses* as a relation among programs. For programs A and B, "A uses B if correct execution of B may be necessary for A to complete the task described in its specification." In other words, A *requires* service that B *provides*. Parnas notes that "A may use B even though it never invokes it." The present model, which does not involve programs, nor their invocations, reinterprets *uses*. Developers explicitly record the uses relation using the *provides* and *requires* relations of the *Module* class. *Provides* indicates resources a module provides that other modules may use. A module's *requires* relation indicates resources that it uses. The resources are recorded by the *Interface* class, which comprises two relations, *input* and *output*, to the *Variable* class. The former indicates variable values that the interface accepts. The latter indicates variable values the interface makes available.

The example in Figure 4, adapted from [9], illustrates the relationship between modules, variables, and uses. Each of the three boxes represents a named module, e.g., *AudibleSignal*, *SystemValue*, and *SystemState*. Assigning variable names to compartments named *requires output* and *provides output* implicitly assigns their value functions to the encompassing modules. Assigning variable names to compartments named *requires input* and *provides input* indicates variable values the module will accept (made available by *provides output* and *required output* compartments, respectively, assigned the same variable names), completing the uses relation, which the solid arrows represent. In the figure, the variable at the tail of the solid arrow depends upon the presence of the variable at its head. Dotted arrows indicate the value flow of variables labeling them. Note that no local variables are identified and that none of the modules is completely represented.

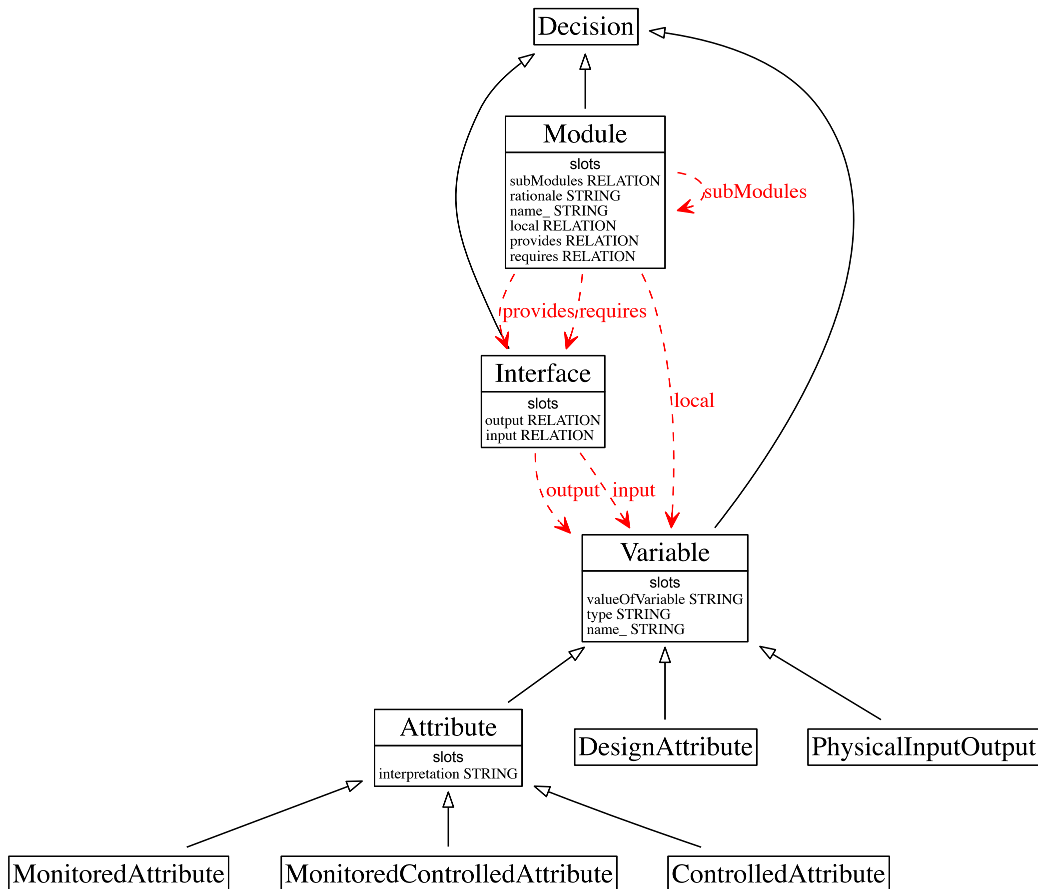

Figure 3 Modularization Decisions

Uses and value flow need not coincide. In the case of the controlled attribute, *cAudibleSignal*, the *AudibleSignal* module is responsible for determining when the signal should sound, which the behavior function associated with *cAudibleSignal* computes. The function depends upon the presence of the provided input of the *SystemValue* module to effect the environment suitably, i.e., sound or silence the alarm. If developers should decide the alarm is unnecessary, *cAudibleSignal* and its behavior function can be deleted, as well as *cAudibleSignal* from the *providesinput* compartment of the *SystemValue* module (since there is only one source for any variable value). In the case of the other two instances of uses, *AudibleSignal* depends upon the presence of the values of *mButtonPressed* and *zSystemMode*, which are in the domain of the behavior function associated with *cAudibleSignal*.

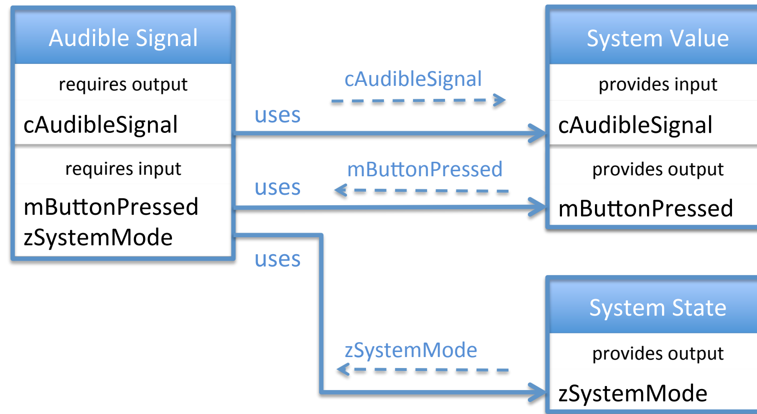

Figure 4 Modules, Uses, and Value Flow

System decisions model

Figure 5 illustrates the ontology of the System Model, which decomposes system behavior into units of execution called *software components*, which interface via exchange of variable values. Top level decisions in this model are recorded by *Environment* class, *Variable* class, and *Interface* class. The System Model shares the *Environment* class and part of its decomposition with the Environment Model, and shares the *Variable* class and its decomposition with the Behavior and Modularization models. What is unique in this model is an elaboration of the *SoftwareComponent* class from the Environment Model and a repurposing of the *Interface* class.

Developers decompose system behavior by assigning variables to the *local* relation of a software component (which is the only component aware of the variable) and to the *output* relation of the *Interface* class of a software component. In each case, the function calculating the value is assigned implicitly to the software component. Values of variables assigned to the *output* relation of the *Interface* class are provided to software components that have those same variables assigned to the *input* relation of their *Interface* class.

PREVIOUS WORK

Previous work developed a related model driven software development approach [8]. Figure 6 illustrates the Sage development method and associated tool chain, which intends to support an incremental, iterative, model-driven process to build and sustain high assurance, reactive multi-agent systems. Sage supports four overlapping models similar to those discussed above: Environmental Model (above called Environment Model), Behavioral Model (Behavior Model), Design Model (Modularization Model), and Run-time Model (System Model). These models comprise the Sage PIM (Platform Independent Model). Sage calls *agents* what this paper refers to as *software components*.

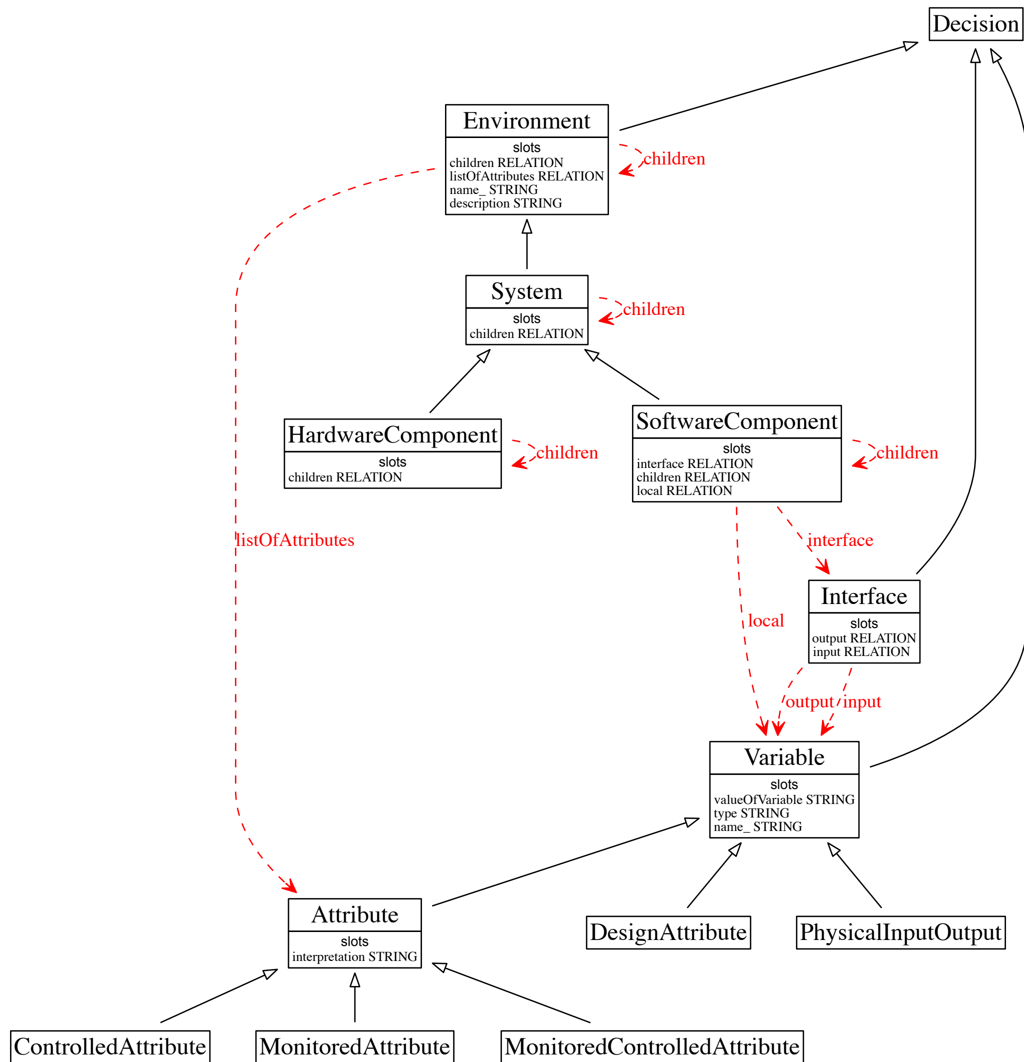

Figure 5 System Decisions

Sage agents are loosely-coupled, location-transparent executable software components that monitor the environment and respond to changes in it. The Sage tool chain includes a model-driven development environment that uses a set of variables and associated tabular functions to specify system behavior, translates the tabular functions to the Sol execution language [1] (creating a PSM—Platform Specific Model), and to Sal, the language of the Salsa invariant checker [2]. The SINS middleware infrastructure [3] provides a distributed execution environment, which implements a topic-based publish/subscribe communication paradigm [5] used by Sage reactive agents.

Of the several applications that have exercised the tool chain, the largest and most complex is the Weapon Control Panel (WCP) of a major weapon platform, disussed in [6]. Operators use the WCP to monitor and prepare weapons for launch. The logic equations in the external interface requirements of a draft of the contractor-developed Software Requirements Specification (SRS) were translated into the tabular executable functions of the Sage behavioral model. The resulting model comprises 258 variables—

108 inputs, 90 outputs, and 60 internal variables—and 150 functions.

Sage was used to create an environmental model of the WCP, consisting of five entities. Three of the entities are panels of switches (monitored attributes) and indicator lights (controlled attributes). Two of the entities are external mechanical assemblies and their associated attributes. Several run-time models of WCP (which allocate computations to Sage reactive agents) were developed using Sage, demonstrating flexibility, loose coupling, and location transparency. Agents were modified by moving attributes between them; associated tabular functions followed automatically. One run-time model was a single, monolithic agent. Another was a six-agent model. Sage generated Sol code for both run-time models. Translating a Sage-generated Sol model of the WCP to Sal—the language of Salsa—allowed Salsa to check the consistency and completeness of the WCP, finding one non-deterministic function. After compilation by the Sol compiler both models have been deployed and executed on SINS middleware, and demonstrated using scenarios adopted from a draft WCP System/Subsystem Specification.

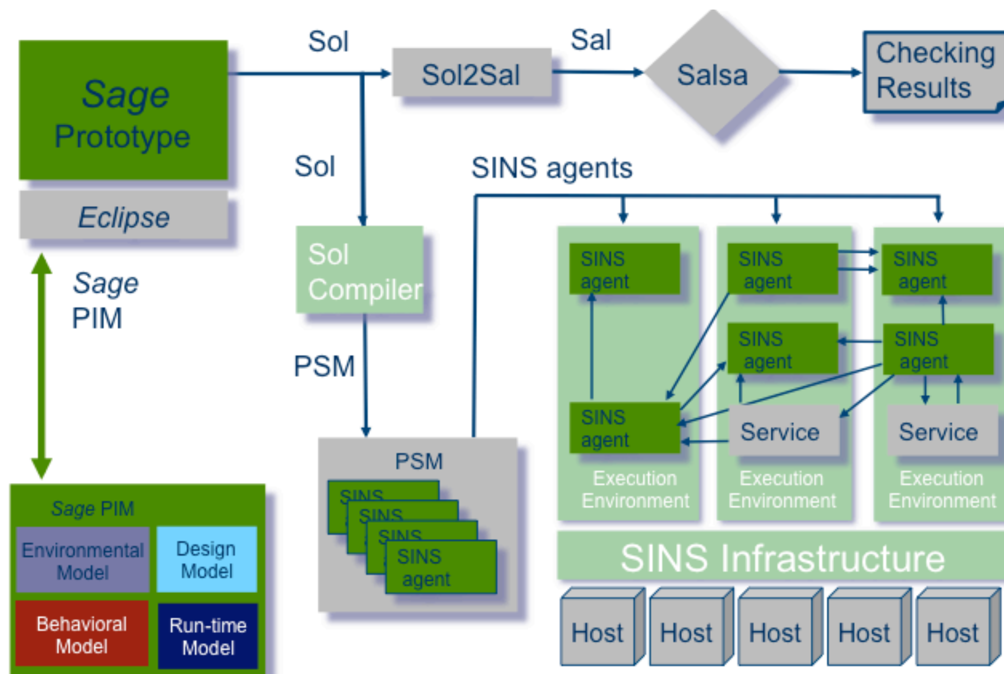

Figure 6 Sage Tool Chain

RELATED WORK

Weiss and colleagues [15] describe a decision-oriented approach to specifying a family of software systems and generating members of that family. A *commonality/variability analysis* records decisions characterizing all members of the family and what distinguishes each member from other members. *Parameters of variation*, which specify the latter, map to a modularized architecture. The architecture itself records decisions regarding how to support generating members of the family. Software modules encapsulate decisions resolving resolution of the parameters of variation, which facilitates composing a family member characterized by assigning them values. A

decision model (different usage of the term than used in other fields) guides the order and timing of decision making implicit in assigning values to parameters of variation.

ACKNOWLEDGEMENTS

The author is grateful to Ramesh Bharadwaj, Grady Campbell, and especially David Weiss for comments that led to improvements in the paper.

REFERENCES

[1] Bharadwaj. R. (2002). SOL: A Verifiable Synchronous Language for Reactive Systems, *Proc. Synchronous Languages, Applications, and Programming (SLAP '02)*, **Electronic Notes in Theoretical Computer Science,** Elsevier, 2002.

[2] Bharadwaj. R. and S. Sims. (2000). Salsa: Combining Solvers with BDDs for Automatic Invariant Checking, in *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000).* **Lecture Notes in Computer Science,** Springer**.**

[3] Bharadwaj. R. (2003). Secure Middleware for Situation-Aware Naval C^2 and Combat Systems, in *Proc. 9th International Workshop on Future Trends of Distributed Computing Systems (FTDCS 2003).*

[4] Britton, K. H., & Parnas, D. L. (1981). *A-7E Software Module Guide* (No. NRL-MR-4702). NAVAL RESEARCH LAB WASHINGTON DC.

[5] Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. *ACM computing surveys (CSUR)*, *35*(2), 114-131.

[6] Heitmeyer, C., Kirby, J., Labaw, B., Archer, M., & Bharadwaj, R. (1998). Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on software engineering*, *24*(11), 927-948.

[7] Heninger, K. L. (1980). Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, (1), 2-13.

[8] Kirby, J. (2006). Model-Driven, Agile Development of Reactive, Multi-Agent Systems, *Proc. 30th Annual International Computer Software and Applications Conference (COMPSAC 2006).*

[9] Kirby, J. (2013). Specifying software behavior for requirements and design. In *Journal of Systemics, Cybernetics and Informatics*, 11(8), 80-88.

[10] Noy, N. F., & McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology, *https://protege.stanford.edu/publications/ontology_development/ontology101.pdf*.

[11] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053-1058.

[12] Parnas, D. L., Clements, P. C., & Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on software Engineering*, (3), 259-266.

[13] Parnas, D., & Clements, P. (1986). A rational design process: how and why to fake it. *IEEE transactions on software engineering*, (2), 251-257.

[14] Parnas, D., & Madey, J. (1995). Functional documents for computer systems. *Science of Computer programming*, 25(1), 41-61.

[15] Weiss, D. M., Li, J. J., Slye, H., Dinh-Trong, T., & Sun, H. (2008, September). Decision-model-based code generation for SPLE. In *Software Product Line Conference, 2008. SPLC'08. 12th International* (pp. 129-138). IEEE.

APPENDIX I COMBINED ONTOLOGY

Ontology of decisions that developers of software-intensive systems make.

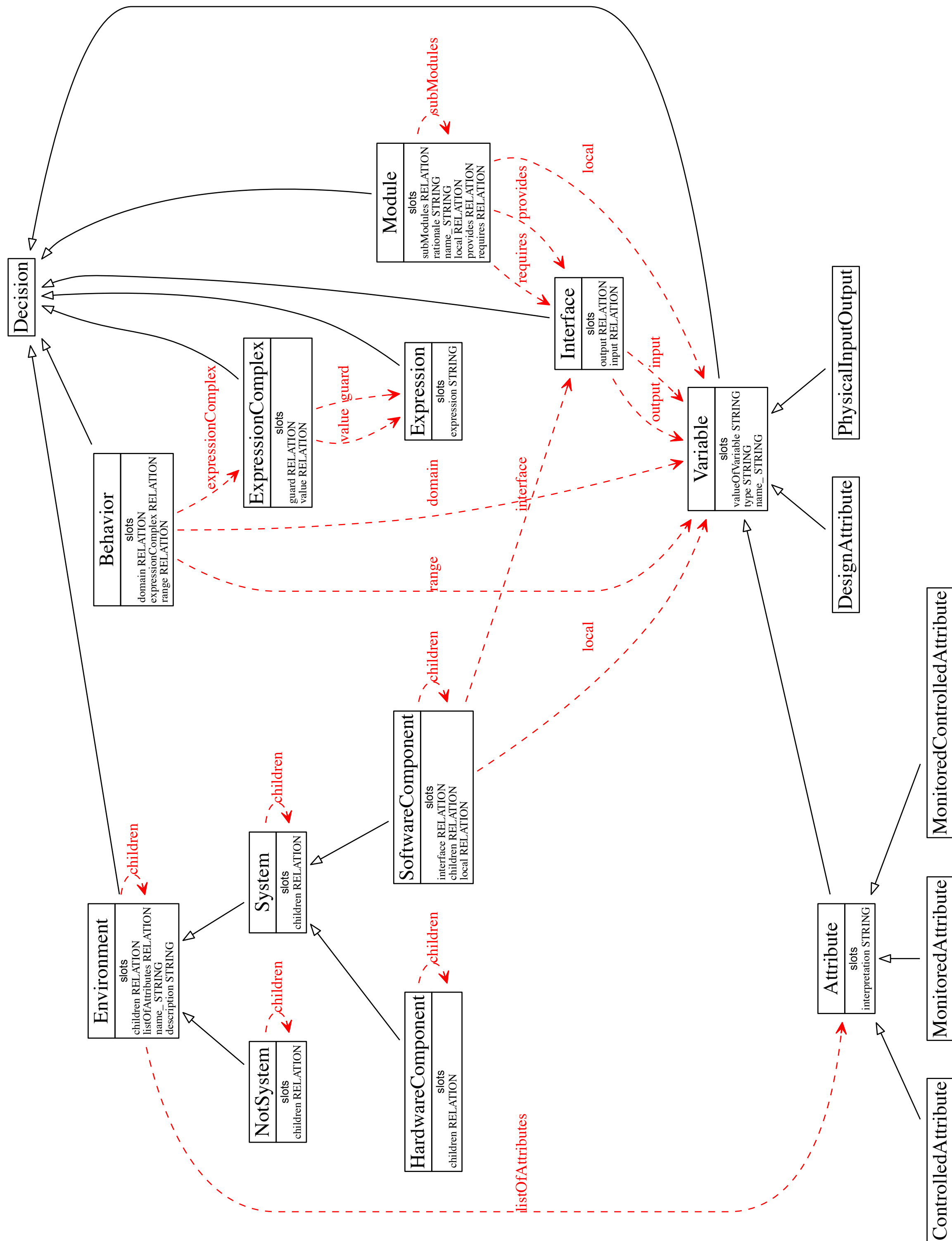

Figure 7 Saj Decisions Ontology

APPENDIX II CLASS DICTIONARY

Definitions of classes representing decisions that developers of software-intensive systems make. Each definition begins with a description of the decision. The *IS-A* line identifies what class the decision specializes. Slot names and their types identify properties of each decision.

Attribute

```
"Value of environmental characteristic."
IS-A Variable
SLOTS ARE:
      interpretation   STRING
```

Behavior

```
"Value of variable over time."
IS-A Decision
SLOTS ARE:
      domain     RELATION
      expressionComplex RELATION
      range RELATION
```

ControlledAttribute

```
"Attribute whose value the system can set, affect, or influence."
IS-A Attribute
```

Decision

```
"Software production decision."
```

DesignAttribute

```
"Variable defined for convenience of specification, description, or
modeling."
IS-A Variable
```

Environment

```
"Thing in environment of software-intensive system."
IS-A Decision
SLOTS ARE:
      children   RELATION
      listOfAttributes RELATION
      name_ STRING
      description STRING
```

Expression

```
"Expression of value."
IS-A Decision
SLOTS ARE:
      expression  STRING
```

ExpressionComplex

```
"Expression of value and when it applies."
IS-A Decision
SLOTS ARE:
      guard RELATION
      value RELATION
```

HardwareComponent

```
"Hardware component of the system."
IS-A System
SLOTS ARE:
           children   RELATION
```

Interface

```
"Shared values."
IS-A Decision
SLOTS ARE:
      output     RELATION
      input RELATION
```

Module

```
"Decomposition of work into units."
IS-A Decision
SLOTS ARE:
      subModules  RELATION
      rationale   STRING
      name_ STRING
      local RELATION
      provides   RELATION
      requires   RELATION
```

MonitoredAttribute

```
"Attribute whose value the system can measure, monitor, or
determine."
IS-A Attribute
```

MonitoredControlledAttribute

```
"Attribute whose value the system can both measure, monitor, or
determine and set, affect, or influence."
IS-A Attribute
```

NotSystem

```
"Thing in environment that is not the system nor part of the
system."
IS-A Environment
SLOTS ARE:
      children   RELATION
```

PhysicalInputOutput

```
"Denotes value sent to or received from physical, cyber, or virtual
device."
IS-A Variable
```

SoftwareComponent

```
"Software component of the system."
IS-A System
SLOTS ARE:
      interface  RELATION
      children   RELATION
      local RELATION
```

System

```
"System or component of the system."
IS-A Environment
SLOTS ARE:
      children   RELATION
```

Variable

```
"Value that changes over time."
IS-A Decision
SLOTS ARE:
      valueOfVariable  STRING
      type  STRING
      name_ STRING
```

APPENDIX III SLOT DICTIONARY

Definitions of slots representing properties of decisions that developers of software-intensive systems. Each definition begins with a description of the property. Each property may be a typed value (indicated by TYPE) or an instance of a class (indicated by RELATION). The comma-separated pair following CARDINALITY indicates the minimum and maximum number of a property, respectively, that may be associated with a decision.

children

```
"Children property."
RELATION Environment
CARDINALITY 0, many
```

description

```
"Statement characterizing thing in environment."
TYPE STRING
CARDINALITY 0, 1
```

domain

```
"Domain property."
RELATION Variable
CARDINALITY 0, many
```

expression

```
"Text of an expression."
TYPE STRING
CARDINALITY 0, 1
```

expressionComplex

```
"ExpressionComplex property."
RELATION ExpressionComplex
CARDINALITY 0, many
```

guard

```
"Guard property."
RELATION Expression
CARDINALITY 0, many
```

input

```
"Input property."
RELATION Variable
CARDINALITY 0, many
```

interface

```
"Interface property."
RELATION Interface
CARDINALITY 0, 1
```

interpretation

```
"How value of attribute relates to quantity or quality that
attribute denotes."
TYPE STRING
CARDINALITY 0, 1
```

listOfAttributes
```
"Attributes property."
RELATION Attribute
CARDINALITY 0, many
```

local
```
"Local property."
RELATION Variable
CARDINALITY 0, many
```

name_
```
"Name of thing."
TYPE STRING
CARDINALITY 0, 1
```

output
```
"Output property."
RELATION Variable
CARDINALITY 0, many
```

provides
```
"Provides property."
RELATION Interface
CARDINALITY 0, 1
```

range
```
"Range property."
RELATION Variable
CARDINALITY 0, many
```

rationale
```
"Rationale for decision."
TYPE STRING
CARDINALITY 0, 1
```

requires
```
"Requires property."
RELATION Interface
CARDINALITY 0, 1
```

subModules
```
"Submodule property."
RELATION Module
CARDINALITY 0, many
```

type
```
"Characterizes type of values that attribute assumes."
TYPE STRING
CARDINALITY 0, 1
```

value
```
"Value property."
RELATION Expression
CARDINALITY 0, 1
```

valueOfVariable

```
"Characterizes values that variable assumes."
TYPE STRING
CARDINALITY 0, 1
```