SPECIFICATION BY EXAMPLE USING
GRAPHICAL ANIMATION AND A PRODUCTION SYSTEM

Richard St-Denis

Département de mathématiques et d'informatique
Université de Sherbrooke, Sherbrooke, QC, CANADA, J1K 2R1

Abstract

This paper describes a rapid prototyping technique based on executing requirement specifications for application domains. Entities, activities, and constraints of a real environment are specified and represented, respectively, by icons, icon transformations, and production rules. The behavior of the system is simulated by executing the production rules combined with an icon animation. The aspects chosen to be modeled are those concerning operational strategies encountered in the application environment. Thus, we show how to organize a knowledge base and build graphic objects from a formal description of an application domain. Furthermore, we briefly present a methodology that allows designing a system in such a way that it includes some parts of its prototype during and after development. The prototyping technique and the design methodology can be gathered together to support the incremental development of large systems.

Introduction

Interest in using a conceptual model of the application domain in the development and maintenance of software systems has created a need for tools to process domain knowledge. Research to date has focused on defining languages based on concepts of knowledge representation, thereby creating new ties between software engineering and artificial intelligence[1,2]. This observation gives rise to two problems: the validation of the conceptual model and its methodical use in every life cycle phase. The main objective of this paper is to provide a coherent solution to both these problems. In precise terms, this means:

- using the traditional software development process augmented by a prototyping cycle[3];
- developing a design methodology that brings out the relationships between domain knowledge and the system's components; and
- designing and building a software validation tool that allows the rapid production of prototypes, partially usable when the system is being implemented.

Our solution is based on the notion of knowledge-driven systems[4] which generalizes that of table-driven systems. The significant importance of such systems is that they comprise heuristic expressed in a declarative language, in addition to data and procedural algorithms. Therefore, knowledge-driven systems embrace three analogous concepts as shown in Figure 1: evolutionary programs (E-programs) composed of stable modules (S-modules) and evolutionary elements (E-elements)[5], the principal of separation of mechanism and policy[6], and the association of a declarative paradigm with the procedural paradigm.

![Knowledge-driven system](image)

Figure 1 – Basic principles to knowledge-driven system development

Knowledge-driven system construction involves designing a model of some part of the world and extracting knowledge with a view to formulating, experimenting with, and evaluating various heuristics. Each heuristic is an executable specification that formally states the assertions and constraints that a given operational strategy must satisfy. Figure 2 illustrates our approach to developing a knowledge-driven system. Details about this process development and problems related to this approach can be found in a companion paper[7].
Our primary goal in this paper is to present the validation technique adapted to this process. It is based on the design of a particular kind of prototype. The entities, activities, and constraints of the application domain are represented, respectively, by icons, icon transformations, and production rules. Prototype execution requires the design and building of a software tool that uses 2D graphics representation techniques to manipulate the icons and an inference engine to simulate their behavior. Such a tool exemplifies a rapid prototyping technique to mimic reality. To demonstrate that our objective of coherence has been met, we must show the correlation between prototypes and knowledge-driven systems. The proof is based on the fact that the set of production rules associated to E-elements in a knowledge-driven system and the set of production rules in its prototype are dual in the sense that they are identical, but the former is executed in backward chaining and the latter in forward chaining.

The following sections introduce a number of related issues about the validation process of domain knowledge. First, we state the reasons for our approach. We also present the CML[8] language, stressing the motivation in choosing this language as a formalization support for the knowledge of an application domain. We then show how to associate a graphical representation and a set of production rules to a CML specification. We illustrate these associations with the help of an example taken from a lift system[9]. We indicate some of the problems of our prototyping technique and describe the chosen solutions. Finally, we show how a prototype may serve in successive versions of the operational system.

Validation of Domain Knowledge

Difficulties in software development are often a result of problem complexity, not just of size. In many cases, heuristics are important because they are operational strategies that evolve continually. So, software systems may quickly become unreliable and inadequate. This leads to pressure for system change[5]. Promoting ease of modification and maintenance of a system requires evaluation of potential changes to operational strategies. This task is itself very complex. To effectively evaluate such changes it is necessary to know the effect of an operational strategy on the system. Researchers believe domain knowledge to be very important because it provides a way for both defining and validating operational strategies[10].

The knowledge base of an application domain encompasses a description of the objects, phenomena, and policies of an organization within which one or more computer systems operate. This description, called a specification of the application domain, must be stated in a formal specification language. The main advantage of formal languages is that they allow verification and validation of specifications by automated techniques[11]. The verification consists of making sure that the specification is complete and consistent in itself. The validation consists of showing that the conceptual model of the application domain correctly reproduces the realities or phenomena of the environment. However, languages in themselves cannot adequately support the validation process. They require extensions to provide constructs for graphical representation, user interaction, and specification execution.
Typically, the formal specification that must be revised during the validation phase should be presented in a manner that the user may choose to examine the model either as a document or as a simulation. Formalisms that naive users generally have difficulties understanding should be abstracted. Furthermore, there should be facilities that give the user the possibility to dynamically observe and evaluate the conceptual model from his own point of view in real situations. If the specification language is executable (or if the specification is easily translated into an executable formalism) and if it includes semantics such that graphic objects can be associated with a formal specification, then the latter becomes a prototype that permits the simulation and visualization of the behavior of the objects of the application domain. This prototype constitutes a workbench that both users and developers can directly interact with, not only for a common understanding of problems, but also as a source for elements of solution. Figure 3 shows the elements addressed in this paragraph with regard to the prototyping cycle.

Figure 3 – More details on the prototyping cycle

Choice and Presentation of the CML Language

The validation environment is centered on the CML[2,8] language in order to represent knowledge of application domains. CML is an object-oriented requirements specification language that provides facilities for the organization, formalization, and management of objects and constraints of a closed universe. We have adopted this language for four fundamental reasons.

1. CML uses knowledge-representation techniques developed in artificial intelligence. It is thus possible to translate a CML specification into an executable formalism that supports problem solution by symbolic evaluation. Since the assertions (i.e., constraints) are objects, CML offers all the power of a logic-oriented approach, in addition to all the advantages of an object-oriented one.

2. The concepts and mechanisms supported by CML facilitate the description of objects and their relationships. The correspondence between components of the real world and those of a conceptual model, expressed in an object-oriented formalism, is natural since the objects and relationships are taken directly from the system we are modeling.

3. The semantics of CML is defined explicitly, which permits removing any ambiguity in the meaning of a specification and clarifying the notion of specification consistency.

4. Relationships are also objects. Therefore, it is possible to define attribute classes according to specific applications. For the purpose of rapid prototyping, attribute classes have been defined such that their associated semantics suggests a well-ordered procedure for translating a CML specification into an executable graphic specification. Figure 4 summarizes the major attribute (or property) classes that we consider in this paper. Many of them are borrowed from RML[12], the predecessor of CML.

![property classes and their semantics](image)

In CML, the relationships between objects are established by using the property concept. A property is an object that allows each object to have its own attributes. This concept is subordinate to the basic component of CML called a proposition. A proposition is a 4-tuple

(source, attribute, destination, duration)

where each element is itself another proposition. It expresses the fact that the attribute name (attribute) associated with the subject (source) has a given value (destination) during a time interval (duration).

239
CML supports three abstraction mechanisms, which can be stated precisely from the property concept. Aggregation uses properties to associate a set of objects to a common subject. Classification groups objects that share common properties into classes (or metaclasses). Generalization permits defining a more general class from the properties shared by several specific classes.

**An Example**

This section presents part of a specification of a lift system written in CML. We have chosen this example because it includes a variety of sophisticated problems. While this system considers only requests from inside the lifts (there are no external buttons on each floor), the example is sufficiently complete to illustrate the most important concepts explored in the following sections.

Figure 5 contains a subset of the entities of the lift system. Due to space limitation, we have omitted specifying some attributes, such as the lift velocity and load, installation constraints, operational conditions, and the definition of the general classes. The instances of the classes LiftSystemEntity and LiftEntity have as their unique property, respectively, a lift system and a lift. The first definition is useful to implicitly state the fact that entities which belong to the class Lift or Floor are related to a specific lift system. Since these two classes are specializations of the generic class LiftSystemEntity, they inherit the attribute "lift system". The second definition plays a similar role. It allows to relate entities which belong to the class Interface, LiftDoor, or InternalRequest to a specific lift. The class Lift defines lift entities. A lift has a passenger interface and a door as components. It also has four attributes (a position, a direction, a state, and one or more requests) and four activities (arrive, pass, stop, and depart) which can modify its attributes.

Figure 6 describes some of the activities of the lift system. The classes LiftActivity and DoorActivity have as instances the activities in which a lift (denoted by the identifier Lift) and a door (denoted by the identifier Door) participate. Each activity class definition state conditions under which the activity starts (actcond) and assertions that are true at the end of the activity (postcond). For example the class definition StopFloor state that a lift must stop if it arrives at the ground, top floor, or a floor for which it has a request. At the end of this activity the lift is stopped and its door unlocked. In general, when a lift arrives at a given floor, it stops or goes on to the next floor. If it stops, it unlocks and opens its doors, closes and locks them, and leaves for another floor. Initially, the doors are open, the lift is at the ground or top level, and its direction is undefined. When one or more internal requests are generated, the lift closes its doors and leaves the ground or top level.

**Figure 5 - Lift system entities**

CLASS LiftSystem in ENTITY-CLASS WITH
  Necessary subsystem
  Lifts:LIFT
  Floors:FLOOR

CLASS LiftSystemEntity in ENTITY-CLASS WITH
  Single link
  Lift-system:LIFT_SYSTEM

CLASS Lift in ENTITY-CLASS
  ISA LiftSystemEntity WITH
  Necessary Single subsystem
  PassengerInterface:INTERFACE
  Door:LIFT_DOOR
  Necessary Single attribute
  Position:GROUND..TOP
  Direction: ("up", "down", "undefined")
  State: ("arrived", "passed", "stopped", "departed")
  Attribute
  Requests:INTERNAL-REQUEST
  Modifier
  Arrive:ARRIVE_FLOOR
  Pass:PASS_FLOOR
  Stop:STOP_FLOOR
  Depart:DEPART_FLOOR

CLASS LiftEntity in ENTITY-CLASS
  ISA LiftSystemEntity WITH
  Necessary Single attribute
  Lift:LIFT

CLASS Floor in ENTITY-CLASS
  ISA LiftSystemEntity WITH
  Necessary Single attribute
  Number:GROUND..TOP

CLASS Interface in ENTITY-CLASS
  ISA LiftSystemEntity WITH
  Necessary subsystem
  Lift-buttons:INTERNAL_BUTTON

CLASS LiftDoor in ENTITY-CLASS
  ISA LiftEntity WITH
  Necessary Single attribute
  State: ("closed", "open", "locked", "unlocked")
  Modifier
  Close:CLOSE_DOOR
  Open:OPEN_DOOR
  Lock:LOCK_DOOR
  Unlock:UNLOCK_DOOR

CLASS InternalRequest in ENTITY-CLASS
  ISA LiftEntity WITH
  Necessary Single attribute
  Number:GROUND..TOP
  Producer
  Push:PUSH_BUTTON
  Consumer
  OpenDoor:OPEN_DOOR

Figure 5 - Lift system entities
CLASS LIFT_ACTIVITY IN ACTIVITY-CLASS WITH
  Single control
  Lift:LIFT

CLASS ARRIVE_FLOOR IN ACTIVITY-CLASS
  ISA LIFT_ACTIVITY WITH
  actcond
  lift.departed_or_passed (Lift.state="departed") Or (Lift.state="passed")
  postcond
  lift.arrived: Lift.state="arrived"
  upper.floor: (((Exists p) (Lift.position=p)) And (Lift.direction="up") => (Lift.position=p+1))
  lower.floor: (((Exists p) (Lift.position=p)) And (Lift.direction="down") => (Lift.position=p-1))

CLASS PASS_FLOOR IN ACTIVITY-CLASS
  ISA LIFT_ACTIVITY WITH
  actcond
  lift.arrived: Lift.state="arrived"
  not_at_ground: Lift.position+GROUND
  not_at_top: Lift.position+TOP
  nosequest: Not ((Exists r) (r ElementOf Lift.requests) And (r.number=Lift.position))
  postcond
  lift-passed: Lift.state="passed"

CLASS STOP_FLOOR IN ACTIVITY-CLASS
  ISA LIFT_ACTIVITY WITH
  actcond
  lift.arrived: Lift.state="arrived"
  a.request ((Lift.position=GROUND) Or (Lift.position=TOP) Or ((Exists r) (r ElementOf Lift.requests) And (r.number=Lift.position)))
  postcond
  lift-stopped: Lift.state="stopped"
  door.locked: Lift.door.state="locked"

CLASS DEPART_FLOOR IN ACTIVITY-CLASS
  ISA LIFT_ACTIVITY WITH
  actcond
  ready_to_depart: Lift.door.state="closed"
  defined_direction (Lift.direction="up") Or (Lift.direction="down")
  postcond
  lift.departed: Lift.state="departed"
  door.locked: Lift.door.state="locked"

CLASS DOOR_ACTIVITY IN ACTIVITY-CLASS
  ISA DOOR_ACTIVITY WITH
  Single control
  Door:DOOR

CLASS CLOSE_DOOR IN ACTIVITY-CLASS
  ISA DOOR_ACTIVITY WITH
  actcond
  door.open: Door.state= "open"
  a.request ((Door.direction= "up") Or (Door.direction= "down") Or ((Exists r) (r ElementOf Door.requests)))
  postcond
  door.closed: Door.state= "closed"
  go.up (Door.lift.position=GROUND) =>
  go.down (Door.lift.position=TOP) =>

Figure 6 - Lift system activities (continued)

The Prototyping Techniques

There are three essential aspects to examine in designing a logical and graphics-oriented validation environment: animation, interaction, and behavior.

1. Animation - Realistic animation is a powerful means of validating the user's specification. It helps both the user and analyst to understand, clarify, and explain the information being displayed by the validation tool because the images represent real objects as well as their spatial and temporal relationships.

2. Interaction - The validation tool must allow the user to interrupt or alter the animation once it has begun. He can then methodically run different test cases on the prototype to validate special situations against incidental events.

3. Behavior - A system can be in any one of a finite number of possible internal states. Therefore, the validation tool must include rules which dictate a state transition logic to reproduce the system evolution step-by-step or event-by-event. In our approach, we use situation-action or if-then rules which are the basis of production systems[13].

The example presented in the previous section shows CML's ability to represent objects and relationships of the application domain, but not convenient
graphical representations. Since a system is by nature dynamic, a visual modeling method must supply not only a static view of the system, but, more importantly, a dynamic view. The static view displays the system at a given time; the dynamic view illustrates the system's behavior, that is, the state transitions occurring within a time interval. In our perspective, the transitions are the result of activities defined over entities. Furthermore, the static view represents a particular case of the dynamic view: it contains a subset of the system's entities, each entity being in a precise state.

To build the static and dynamic views of the system, the validation tool manages three windows: the icon definition window, the icon transformation definition window, and the animation window.

In the icon definition window, the user creates icons from templates (e.g., broken lines, curves, geometric figures, predefined icons) and primitives (e.g., superimposition and juxtaposition of elementary icons, filling) similar to those found in an interactive graphics package. Some of these primitives are based on abstraction mechanisms supported by CML. For example, the juxtaposition and superimposition of icons mirror the aggregation and specialization mechanisms. Therefore, icons are considered as three-dimensional objects because they inherit the graphic properties associated to their subentities as well as those of their superclasses. A set of icons is associated with each entity class of a CML specification. All the icons in this set represent the same entity but in different states. They are necessary to define the icon transformations associated with the activities. Examples of entities and their graphical representation are shown in Figure 7.

An instantiation of an entity class is produced each time an icon goes from the icon description window to the animation window. Normally, the set of icons placed in the animation window constitutes a static view which must represent a consistent state of the system. The consistency of a static view must be verified before the starting of an animation. Note however, that the spatial relationships between the icons are not, in general, included in the constraints of the conceptual model. The definition of these relationships is implicit and the user's responsibility during the displacement of the icons in the animation window. As an example, Figure 8 depicts the animation window for a system with two lifts as well as a part of facts associated to this static view.

![Figure 7](image1.png)  
**Figure 7** - Graphical representation of lift system entities

![Figure 8](image2.png)  
**Figure 8** - Example of an animation window
In the icon transformation definition window, the user produces a set of graphic transformations for each activity of a CML specification. For each transformation, one indicates a starting and ending icon, and one or more visual effects:

starting icon \[\xrightarrow{\text{visual effects}}\] ending icon

If the ending icon is not specified, the starting icon is removed from the animation window. Conversely, if the starting icon is absent then the ending icon is displayed on the screen. Figure 9 illustrates some icon transformations for the lift system.

In order to be able to specify the visual effects, the software tool embodies various primitives for reproducing the behavior of the objects of the system in a manner as consistent and natural as possible (e.g., translations, rotations, progressive scale transformations, movements along a trajectory, color intensity changes). We have preferred this solution to the one consisting of defining a set of intermediate states so as not to overload the user's work in the construction of the icons. The visual effects apply to the icon or to a group of elementary parts of the icon. The user must render consistent the visual effects of the transformations and the spatial relationships between the icons in the animation window.

The icon transformations represent images in an animation sequence. The dynamic view of the behavior of the objects comes from the successive firing of the production rules which contain in their consequent part the calls to the graphics package. Figure 10 shows an example of an animation sequence.

Execution of a Specification

The programming environments developed for artificial intelligence applications constitute popular tools to implement prototypes rapidly and economically. However, the implementation of a knowledge base as an aid in properly simulating the environment of the application domain necessitates taking the following into consideration:

1. the form and type of the assertions;
2. the modification of the base of facts; and
3. the order of matching and firing of the rules.

Rule Base

The assertions of a CML specification expressed in first order logic must be grouped and converted into clausal form [14] to identify the condition-action pairs of the production rules. The rules are grouped into two general contexts: a context of simulation of the behavior of the objects and a context of consistency verification. The verification rules have priority over the simulation rules: if they are triggered, they will be fired before the simulation rules.

For a given activity, the simulation rules come from the following proposition:

\[F_{\text{actcond}} \rightarrow F_{\text{postcond}}\]

where \(F_c\) is the conjunction of the assertions of property class \(c\) (actcond or postcond) of the activity.
Furthermore, actions must be added, depending on the case, to delete or create all the facts corresponding to each entity of property class input or output. Each simulation rule associated with an activity and obtained from the conversion procedure must represent the activity in its entirety (i.e., the activities are atomic). Finally, in the action part of a simulation rule, we insert as many calls to the graphics package as there are icon transformations associated to the corresponding activity. Figure 11 gives some simulation rules written in a SNARK[15] dialect. This language is based on first-order logic. Each rule has two parts separated by the delimiter "- >". The first part specifies the actual conditions under which the rule can be fired and the second part specifies actions or postconditions (changes to the knowledge base). A symbol between brackets denotes a variable and "< -" an assignment. Expressions are written in prefix notation.

The verification rules are obtained from the following proposition:

\[ F \text{actcond} \land (\neg F \text{activity}) \land (\neg F \text{entity}) \rightarrow \text{inconsistency} \]

where \( F \text{actcond} \) is the conjunction of the assertions of property classes precond and constraint of the activity and \( F \text{entity} \) is the conjunction of the assertions of property classes initcond, finalcond and constraint of the entities which are, respectively, produced (output), consumed (input) or modified (control) by the activity.

It should be noted that we model the state transitions by specifying the conditions under which an activity starts and the conditions which are true after the execution of the activity (i.e., simulation by activities compared with simulation by events). Furthermore, the verification rules take into account the logical constraints that must be satisfied between each transition during the execution of the model.

**Base of Facts**

Initially, the base of facts is created from the icons located in the icon animation window and from the values of the properties of their corresponding entities. During animation, the user can add, change, or delete facts by using programmable keys or by pointing to an icon with the mouse and changing its property values. Adding or deleting facts corresponds to input/output of data normally accepted by the system (e.g., pushing a button). The modification of the base of facts must not challenge the consistency of the state of the system.

**Firing of the Rules**

The order in which the icon transformations are executed is obtained implicitly, on the one hand, by simulation rules which specify the transitions between the activities, and, on the other hand, by a mechanism of alternating execution of the graphic processes. These two steps, however, are not sufficient to ensure realistic animation. We must take an additional precaution to guarantee an order in the firing of rules which would be fair for all instances of system objects.

RULE { 1.1 depart floor }
STATE(Door([LIFT]))=closed
DIRECTION([LIFT])=up
- >
STATE([LIFT])< -departed
STATE(Door([LIFT]))< -locked
EXECUTE([LIFT],depart-up)
RULE { 2.1 pass floor }
STATE([LIFT])=arrived
DIRECTION([LIFT])=up
not REQUESTS([LIFT])=[REQ]
FLOORNUMBER([REQ])=POSITION([LIFT])
not POSITION([LIFT])=top
- >
STATE([LIFT])< -passed
STATE(Door([LIFT]))< -unlocked
EXECUTE([LIFT],pass-up)
RULE { 3.1 stop floor }
STATE([LIFT])=arrived
REQUESTS([LIFT])=[REQ]
FLOORNUMBER([REQ])=POSITION([LIFT])
DIRECTION([LIFT])=up
not POSITION([LIFT])=top
- >
STATE([LIFT])< -stopped
STATE(Door([LIFT]))< -unlocked
EXECUTE([LIFT],stop-up)
RULE { 3.2 atop floor }
STATE([LIFT])=arrived
POSITION([LIFT])=
DIRECTION([LIFT])=up
not
POSITION([LIFT])=top
- >
STATE([LIFT])< -stopped
STATE(Door([LIFT]))< -unlocked
EXECUTE([LIFT],atop-up)
RULE { 4.1 arrive floor }
STATE([LIFT])=departed
DIRECTION([LIFT])=up
POSITION([LIFT])=[p]
- >
STATE([LIFT])< -arrived
POSITION([LIFT])< - [p] 1
RULE { 4.2 arrive top }
STATE([LIFT])=passed
DIRECTION([LIFT])=up
POSITION([LIFT])=[p]
- >
STATE([LIFT])< -arrived
POSITION([LIFT])< - [p] 1

Figure 11 - Example of simulation rules
First, we can use a set of production rules and an inference engine for each instance of an entity class. The inference engines are executed concurrently and communicate with each other through the use of messages for updating their base of facts. Second, we can use a set of production rules for each instance with a single inference engine which executes candidate rules with the same priority. Finally, we can use a single set of production rules and a single inference engine which produces facts using a depth-first technique. In any case, the inference engine works in forward chaining. Furthermore, meta-rules can be used to express preferences over candidate rules.

Relation between Prototypes and Knowledge-driven Systems

The model presented in the introduction does not favour particular techniques or methods for the requirement specification and design phases. However, the specification design resulting from the application of a methodology must contain not only a description of the architecture but also explicit relationships (called K-relations) between its components and domain knowledge as well as design-related decisions. Particularly, its application requires identifying connection points. A connection point refers to E-elements which can be implemented by using production rules obtained during the construction of the prototype. Therefore, a link is established between each connection point and some portion of domain knowledge. These ideas have been recently applied to JSD[10] methodology[17] in which an entity is modelled by showing the ordering of its activities with the aid of a structure diagram notation. The design specifications are augmented-logical structures, which represent a sketch of the architecture combined with K-relations. Augmented-logical structures must be simplified to produce logical structures that will be delivered to the detailed design phase. Figure 12 shows the logical structure of a lift. The connection point P specifies if a lift must go on or stop at the current floor. It represents a problem that can be solved by using rules 2.1, 3.1, and 3.2 in Figure 11.

Conclusion

The problem of validating domain knowledge has been considered. We have shown how, by investing few resources, one can build a tool to validate a description of an application domain. Efforts have been put on mechanisms that increase realism during the simulation of the conceptual model. A working prototype of the validation tool has been implemented on a Macintosh to demonstrate the viability of the method proposed in this paper. Currently, the working prototype is being enhanced to further increase realism by exploiting parallel computing, and to allow a user to interactively browse or change information through an object navigator.

Because our prototyping technique has been designed in a software engineering context by adopting techniques from artificial intelligence, it could be used in the evolutionary process of knowledge system development for validating knowledge of expert systems. However, the specific solution described here is limited. An extension to distributed and real-time applications is currently under investigation. In such applications temporal constraints, performance parameters and response time are important and require specific support.

References


Acknowledgments

We would like to thank Jean Goulet for his assistance on an earlier draft of this paper. We would also like to thank the referees for their comments and suggestions.