From Metamodels to Models: Organizing and Reusing Domain Knowledge in System Dynamics Model Development

MÁRCIO DE OLIVEIRA BARROS
CLÁUDIA MARIA LIMA WERNER
GUILHERME HORTA TRAVASSOS

COPPE / UFRJ – Computer Science Department
Caixa Postal: 68511 - CEP 21945-970 - Rio de Janeiro – RJ
Fax / Voice: 5521 590-2552
{marcio, werner, ght}@cos.ufrj.br

Abstract

System dynamics models can become very complex as they grow. When a model spans through several distinct elements within a certain domain, a large number of equations are needed to describe their detailed behavior and interactions. Large models are hard to understand and develop. However, small models may not provide the desired level of detail in several applications. There is the need for techniques that enhance our capacity to develop and analyze complex models.

In this paper we describe an extension to the system dynamics modeling that allows the development and specialization of domain models. Such models provide a high level representation for future developers within the domain. Our approach divides system dynamics model development in three steps. First, an expert in a given domain develops a model, which conveys the relevant categories of elements that compose the domain and the relationships among these elements. Next, a developer uses such model to describe a particular problem, by specifying how many elements of each category exist in the model of interest and the particular characteristics of each one. Finally, the model is translated to system dynamics constructors in order to be simulated and analyzed. An application of the proposed approach is also presented.

KEYWORDS: domain modeling, process modeling, metamodel, knowledge reuse

1 Motivation

System dynamic models intend to convey representations for real-world elements. Traditionally mathematical equations based on four constructors of the system dynamics modeling language – stocks, rates, processes, and tables – are used to represent such models. Although these constructors allow representation flexibility, the comprehension of large and complex models become a hard task. Real-world elements are not easily identified in a maze of system dynamics constructors. Their representation is usually spread among several equations, which forces developers to analyze them to determine the precise group of equations that describe the behavior of an element and its relationships to other elements.

Usually, models intend to describe specific problems within a problem domain. However, generic and reusable domain knowledge is commonly embedded within the model equations. This characteristic leads to some limitations to the traditional modeling approach such as:

1. It is harder to distinguish between domain and particular problem characteristics;
2. It inhibits the creation of complex domain models, since each new modeler must acquire, learn, and represent the same domain knowledge within his model, and;
3. It does not provide economy of scale when modeling several problems for the same domain, since every model must repeat domain knowledge modeling.

Besides, system dynamic models tend to describe uniformly all elements pertaining to the same category in the modeling domain. Models usually use average values for such elements' properties. We assume this simplification is due to the system dynamics inherent incapability to describe element properties, since they should be independent model variables, requiring too many equations to be specified.

While we need models that are simple to understand, we also want models that can represent the details of their interacting elements. This perception highlights the need for techniques to enhance the development of complex models. If domain knowledge is clearly separated from the particular problem information, every model developed for that domain can reuse it. By reusing domain information created and organized by previous models, the cost of developing new models within a domain can be reduced.

We propose an approach that handles model complexity by raising the abstraction level of their constructors. By raising the abstraction level we mean that modeling constructors shall represent concepts closer to real-world elements (elements of the problem domain) rather than mathematical postulations (elements of the solution domain). Equations are required for model simulation and analysis, but they are not best suited for model description, since they represent concepts far from the model user problem domain. The model shall be expressed in a language that is closer to the user, being later translated to mathematical representation. So, instead of building a model from system dynamics constructors, model developers reuse domain knowledge, which was previously described by domain experts. Such knowledge forms the constructors for the model developer, which will describe models from problem domain concepts rather than with system dynamics basic constructors.

To raise modeling abstraction level has been demonstrated to increase development productivity and product quality in other knowledge areas that explore models. These areas have demonstrated that, model readability and understandability can be increased. Pressman (1997) presents some experimental results for the software development area. For instance, modern object-oriented software development techniques suggest that when several software applications are to be developed for a particular domain, the development team should concentrate in modeling domain concepts and relationships before developing the first application. Such domain model aims at knowledge sharing and reuse. This approach has been explored by recent software engineering environments (Oliveira et al., 1999) and in the context of domain engineering techniques (Braga et al., 1999). However, software domain modeling has other features than those presented in this paper, such as context analysis, variation points, features, and so on.

A system dynamic metamodel and a translation process compose our approach. Both support domain models construction and reuse. The metamodel is a high level representation for system dynamics models. A language allowing the description of categories of elements that collaborate within a problem domain and their relationships represents the metamodel. The translation process compiles the metamodel representation into system dynamics constructors, which can be used for simulation and model analysis. Model behavior is expressed using extended system dynamics constructors, which are separately described for each distinct element category that composes the domain model.

This paper is organized in five sections. The first one comprises this motivation. Section 2 describes the proposed system dynamics metamodel. Section 3 shows an example of the

proposed techniques. Section 4 provides a comparison between the proposed metamodel and PowerSim object-oriented system dynamics extensions. Finally, section 5 presents some considerations and future perspectives of this work.

2 A Proposed Framework for Complex Model Development

This section presents the proposed system dynamics metamodel and process that allows domain knowledge modeling and reuse. To support the definitions and propositions in the following subsections, we use the software project management domain and a small software project as an example. A simplified analysis of the software project management domain highlights three major categories of collaborative elements: developers, activities, and artifacts. Developers accomplish activities, which produce artifacts, eventually consuming other artifacts. Every software project resembles this pattern of interaction, each project with its particular artifacts, developers, and activities[i].

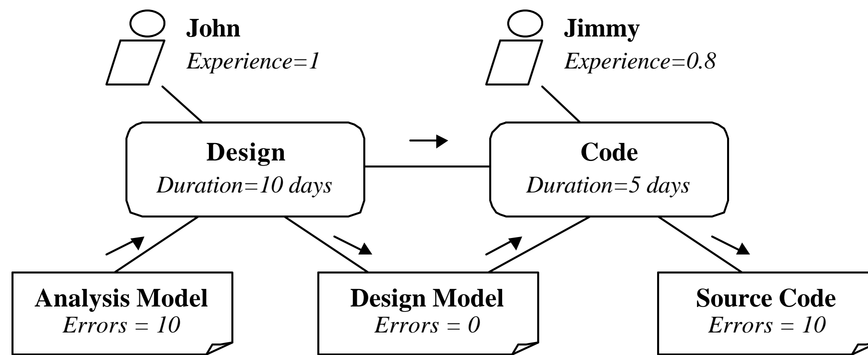

Figure 1 – Small software project example

Consider a particular software project, as depicted in Figure 1. It shows artifacts flowing through the activities and the developers participating in each one of them. The lines in Figure 1 present the relationships among domain elements. Small arrows near the lines represent the direction of the relationship. In this case, two developers, namely John and Jimmy, design and code a software module. The designing activity uses the analysis model as an income to produce the design model. The coding activity uses the previously created design model to produce the software module source code. Developers have experience, shown as a rate in the unity interval, which influences their work quality. Activities have a projected duration, which specifies how many days developers are expected to work in order to accomplish it. Finally, artifacts carry on latent errors, produced by the activities and not corrected by them. Artifacts produced from erroneous incomes shall also contain errors. The relevant information about each element is presented in Figure 1.

2.1 Definitions

This subsection defines the concepts used by the system dynamics metamodel and modeling process. They are referenced throughout the remaining sections of this paper.

A **class** represents a set of elements that can be described by the same properties and exhibit similar behavior. For instance, in the software project model showed in Figure 1, a class describes the whole group of developers, while each particular developer is an element, an **instance** of the class. John and Jimmy are developers' instances.

A class defines the properties that describe its instances. A **property** is relevant information about the class that can assume independent values for each instance. Each particular instance of a class assumes a distinct value for each property defined in the class, depending on the characteristics of the real-world element represented by the instance. In the

software project example, developer experience is a property of the class that represents developers. John and Jimmy have different values for the same property.

A class also defines the behavior of its instances. The **behavior** of a class is a mathematical formulation of its responses to changes in other class instances or in the environment. Such behavior can depend on class properties, allowing distinct class instances to react differently based on their particular characteristics or state. System dynamics constructors describe class behavior.

Class instances can have relationships to other class instances. A **relationship** represents a structural connection between two or more class instances. Such relationships can occur among instances of different classes or instances of the same class. The later is also called an **auto-relationship**.

A **role** represents the part an instance undertakes in a relationship. It denotes the responsibilities and expected instance behavior. In the software project example, the design model artifact plays the role of an income in its relationship with the coding activity.

Finally, a **domain model** contains the classes of elements that cooperate within a problem domain, describing their properties, behavior, and relationships among their instances. The domain model does not describe a model for a specific problem, but a knowledge area where modeling can be applied. It is a generic description of the domain, which should be specialized to a particular problem. In the context of the example, the domain model would capture classes to represent activities, artifacts and developers together their relationships.

2.2 The System Dynamics Metamodel

To allow the development of domain models without specifying specific model characteristics, we propose to use a high level representation, namely a metamodel for system dynamics. Domain experts are able to build domain specific modeling languages using the metamodel language. By doing so, we expect modeling becomes easier than using pure system dynamics constructors, since model developers will use domain concepts described by the domain specific language to build their models. Similar approaches have been used in other areas, for instance the domain model description languages found in (Neighbors, 1981).

The system dynamics metamodel allows the description of domain classes and their relationships. These classes are used as high-level constructors for models developed within the domain. Any model developer interested in describing a problem within that domain specifies the problem in terms of such classes. The developer reuses the behavior described for the classes in the domain model.

For instance, without using the metamodel, the simplified domain showed in Figure 1, would have to be modeled directly in system dynamics equations. However, using the system dynamics metamodel, a software project expert could have a domain model conveying the classes that compose a software project, the behavior of these classes, and their relationships. For this specific situation, activities, artifacts, and developers can be created to describe the problem. The relationships among class instances, such as the precedence relationship among two activities, the accomplishment relationship among activities and developers, and the production and consumption relationships among activities and artifacts, can also be captured.

Model developers can reuse domain models created by experts, describing the specific instances of the domain classes for the problem at hand. Next, developers would specify instances properties values and define their relationships, following the relationships described in the domain models. If the value of a property is not specified for a specific instance, a default value, as specified in the domain model, is assumed.

Regarding the property values in the software project example, every class instance has different property values (i.e., activity duration, developer experience, and so on). So, every

instance property must be represented by an independent equation. Several equations are required to represent the whole set of instances, capturing their particular properties. This leads to larger and error-prone models. By using the metamodel, the model developer just defines the individual property values for the instances and the model behavior would adjust to these new values. Such adaptation does not require structural redesign and no model equations have to be analyzed.

The metamodel allows two types of relationships: multi-relations (also known as 1:N associations), where one instance of a source class is associated to several instances of a target class, and single-relations (also known as 1:1 associations), where one instance of a source class is associated to a single instance of a target class. The model behavior, which can reference such relationships, will be automatically adjusted to the configuration of instances and connections. Such references can be used to define an instance behavior according to the behavior of the instances to which it is associated. Consider that, after observing the model behavior, a project manager decides to try different allocations of the staff among the activities. The model written without using the metamodel would have to be deeply changed for each staff allocation analysis, since the relationships between developers and activities are hard-coded within system dynamics equations. By using the metamodel, the developer would just have to change such relationships, which are clearly stated for each model instance.

A relationship between two or more instances allows one instance behavior equation to assess and even modify other instance behavior (in this case, rate equations affecting foreign stocks). Relationships are unidirectional by default, that is, only the source instance has access to the target behavior. Relationships can be set bi-directional by specifying a role for the target instance. Through this role name, the target instance can manipulate the behavior of its source instance. Auto-relationships are always required to be bi-directional.

The metamodel limits references across relationships though visibility constructors. An instance can access its related instances' information only whether this information is placed in a public area. Private information can be used only by other behavior equations within the same metamodel instance. Table 1 presents a simplified BNF[ii] syntax for the system dynamics metamodel, which is represented as a modeling language. Reserved words are quoted, while non-terminals are presented in italics.

model	= "MODEL" *model_name* "{" { *model_item* } "}" ";"
model_item	= (class | relation) ";"
class	= "CLASS" *class_name* "{" { *class_item* } "}"
class_item	= (*property* | *behavior* | "public" | "private") ";"
property	= "PROPERTY" *property_name default_value*
behavior	= (*stock* | *rate* | *proc* | *table*)
stock	= "STOCK" *stock_name initial_level*
rate	= "RATE" "(" *affected_stock_name* ")" *rate_name rate_expression*
proc	= "PROC" *proc_name proc_expressions*
table	= "TABLE" *table_name table_values*
relation	= (*multi_relation* | single_*relation*)
multi_relation	= "MULTIRELATION" *relation_name source_class* "," *target_ class* [*target_role*]
single_relation	= "RELATION" *relation_name source_class* "," *target_class* [*target_role*]
target_role	= "(" *role_name* ")"

Table 1- System dynamics metamodel simplified BNF syntax

Table 2 presents a simplified BNF syntax for models created using the system dynamics metamodel.

model_instance	= “DEFINE” *model_instance_name model_name* “{” {*class_instance*} “}” “;”
class_instance	= *class_instance_name* “=” “NEW” *class_name* {*definition*}
definition	= (*property* | *relation*) “;”
property	= “SET” *property_name* “=” *property_value*
relation	= “LINK” *relation_name instance_list*

Table 2 – Simplified BNF syntax for models built using the system dynamics metamodel

2.3 The Modeling Process

This subsection presents the proposed system dynamics modeling process based on the definitions presented in section 2.1. We divide the modeling process in three steps. First, an expert in a given domain builds a domain model. Such model cannot be simulated, since it does not specify how many instances of each class exist in the model nor specifies any value for their properties. This step is called **domain modeling**.

The creation of a model from a domain model is the second step in the modeling process, when a model developer specifies how many instances of each class defined for the domain exist in the model of interest. The developer also specifies the instances properties values describing how they are related to each other, based on domain model defined relationships among classes. The resulting model conveys only information about the elements that it uses. It does not present any system dynamics constructor. Such constructors are carried from the behavior of the classes, which are described in the domain model. Behavior equations can be parameterized by the values of the properties of each individual instance, generating different behavior for elements with distinct characteristics. This step is called **model instantiation**.

Finally, the model is translated to system dynamics equations in the third step of the modeling activity. This allows the model to be simulated and analyzed in standard system dynamics simulators. The resulting model uses only standard system dynamics constructors, while the preceding model is described in a high level representation. The high-level representation helps model development and understanding, simplifying the interaction between developers and models. The representation based on system dynamics constructors allows simulation and behavior analysis. This step is called **model compilation**.

3 An Application of the System Dynamics Metamodel

This section shows an application of the system dynamics metamodel and modeling process presented in section 2. We use the software project management knowledge domain and the simplified software project presented in section 2 to show how to develop the domain model for a specific problem, how to create a model from it and how to compile the model to system dynamics constructors.

3.1 Domain Modeling

To exemplify the development of a domain model, consider the project management domain. For the purpose of this example, consider that the relevant classes within the domain are the activities, developers, and artifacts. Developers accomplish activities, which create artifacts, eventually consuming other artifacts. Table 3 presents a simplified model for the project management domain using the proposed system dynamics metamodel language. The model in Table 3 does not specify any behavior. It only shows the syntax for class declaration within the domain model.

```
MODEL ProjectModel
{
	CLASS Developer
	{
		PROPERTY experience 1;
	};
	CLASS Artifact
	{
		PROPERTY latent_errors 0;
	};
	CLASS Activity
	{
		PROPERTY duration 0;
	};
};
```

Table 3 – Simplified model for the project management domain

The MODEL keyword introduces the domain model, namely *ProjectModel*. The model contains three classes, each one declared using the CLASS keyword. The classes are declared within the domain model context, delimited by angled brackets. Each class contains its own context, also delimited by angled brackets, where properties and behavior are declared.

The PROPERTY keyword specifies a property for a class. When using the domain model to create a particular model, a developer can specify the value of the properties for each instance of the classes defined in the domain model. If the value of a property is not specified for an instance, such property assumes a default value, which is specified next to the property name in the domain model.

The project management domain model defines the *Experience* property for the *Developer* class. Such property represents the knowledge and development experience of a developer, which will affect his ability to accomplish activities and capacity to avoid errors. When developing a particular model from the domain model, the developer must determine how many developers are needed and specify each developer experience level. So, the metamodel allows the precise specification of individual property values for each instance of a class. If such precision is not required, an average value can be defined through the property default value, which is used when particular values for each element property are not defined.

The *Artifact* and the *Activity* classes convey a single property each. The *Latent_Errors* property specifies the number of error expected in a previous developed artifact used as the income for an activity. To the purpose of this example, the model assumes that the latent errors in an income artifact are propagated to the artifact produced within an activity. The *Duration* property specifies how much time a group with the same number of average experienced developers needs to accomplish an activity.

Table 4 presents a more complete version of the domain model, containing the relationships among classes. The RELATION keyword represents a single relationship, that is, a relationship with only two participant class instances. The domain model conveys only one single relationship, the *Outcome* relationship, which occurs between one activity and one artifact. It denotes that an activity produces only one artifact. When creating a model from the domain model, a developer must indicate the artifact produced by each activity.

The MULTIRELATION keyword represents a multiple relationship, where one instance of the source class is associated to several instances of the target class. The *Team* relationship, which represents the developers assigned for an activity, is a multiple relationship. There may be several developers assigned for the same activity. The *Team* relationship is also a unidirectional relationship, where only the *Activity* class instance has access to the information about its developers. There must be a role specified for the target

class (in this case, the developers) to make the relationship bi-directional. The role is specified within parenthesis, next to the name of the target class. The *Income* relationship is similar to the *Team* relationship, being a multiple and unidirectional relationship. It represents the artifacts used as income to produce the outcome artifact of the activity.

```
MODEL ProjectModel
{
        CLASS Developer
        {
                PROPERTY experience 1;
        };
        CLASS Artifact
        {
                PROPERTY latent_errors 0;
        };
        CLASS Activity
        {
                PROPERTY duration 0;
        };
        MULTIRELATION Precedence Activity, Activity (NextActivities);
        MULTIRELATION Team Activity, Developer;
        MULTIRELATION Income Activity, Artifact;
        RELATION Outcome Activity, Artifact;
};
```

Table 4 – Model for the project management domain representing class relationships

The *Precedence* relationship is an auto-relationship, since it links instances of the same class. It is also multiple and bi-directional. The *NextActivities* role is specified for the target class, as required by the metamodel. The target class behavior can use the role names to access information about the associated instances. Table 5 presents a complete domain model, containing classes, properties, relationships, and behavior descriptions.

Observe that the domain model behavior equations are distributed among several classes, each class containing its specific behavior. The *Developer* and *Artifact* classes have very simple behavior. The *Developer* class only defines a process to store its *Experience* property value. This allows other instances to consult the property value, since the property itself can only be accessed by its containing instance. The *Artifact* class contains a single stock to represent the expected number of errors in the artifact. Errors could happen during the activity accomplishment and due to latent errors in artifacts used as incomes to produce the current artifact. Since errors are generated during the activity accomplishment, the error generation behavior is located within the *Activity* class.

The *Activity* class contains most behaviors of the domain model. The *TimeToConclude* stock describes the time required to accomplish an activity, being depleted as the simulation advances. The *Work* rate is responsible for depleting this stock. Observe that the stock name, presented within parenthesis after the RATE keyword, associates the rate to the stock. In the metamodel, rates are always supposed to raise their associated stock level. To allow stock depletion, the rate equation must generate negative numbers, as it occurs in the *Work* rate.

For the purpose of this example, an activity can only be executed when all preceding activities are concluded. So, the *Work* rate depends on the *DependOk* process, which determines if the preceding activities of an activity are already concluded. Such process uses the GROUPSUM operator, which sums the values of a selected property for every instance associated to the current instance through a specific relationship. In the *DependOk* process, the GROUPSUM operator sums the level of the *TimeToConclude* stock for every activity that

must be executed before the current one. The *DependOk* process verifies if the operation result is near to zero, determining if the activities have already been accomplished.

```
MODEL ProjectModel
{
	CLASS Developer
	{
		PROPERTY experience 1;
		PROC ExperienceLevel experience;
	};
	CLASS Artifact
	{
		PROPERTY latent_errors 0;
		STOCK Errors latent_errors;
	};
	CLASS Activity
	{
		PROPERTY duration 0;
		STOCK TimeToConclude duration;
		RATE (TimeToConclude) Work if(DependOk, -Min (ExpLevel * TimeToConclude / DT, ExpLevel), 0);
		PROC DependOk GROUPSUM (Precedence, TimeToConclude) < 0.001;
		STOCK ExecutingOrDone 0;
		RATE (ExecutingOrDone) RTExecuting if (AND(ExecutingOrDone < 0.001, DependOk), 1, 0);
		RATE (Outcome.Errors) ErrorsTransmit if (RTExecuting>0.001, GROUPSUM(Income, Errors) / DT, 0);
		PROC ExpLevel GroupMax (Team, ExperienceLevel);
		PROC ErrorsPerDay 5;
		RATE (Outcome.Errors) ErrorsCommited -1 * ErrorsPerDay * Work / ExpLevel;
	};
	MULTIRELATION Team Activity, Developer;
	MULTIRELATION Precedence Activity, Activity (NextActivities);
	MULTIRELATION Income Activity, Artifact;
	RELATION Outcome Activity, Artifact;
};
```

Table 5 – A simple, although complete model for the project management domain

The next two constructors, *ExecutingOrDone* and *RTExecuting*, are used to create a variable that contains zero most of the time, but raises to one in the simulation step that marks an activity start. This variable is used by the *ErrorsTransmit* rate, which raises the number of errors in the produced artifact. In the example, we assume that all errors latent in the income artifacts will be reproduced in the outcome artifact. Although this assumption is not appropriate for most project management models, it is used here to present a mechanism named multirate, which allows a rate to affect several stocks at a time.

A multirate is a rate that can be connected to several stocks through a relationship. If the name of the stock affected by a rate is composed by the name of a relationship, followed by a dot and a stock name in the target class of the selected relationship, the stocks of each associated instance will be affected by the rate. If the relationship is a single one, only one stock will be affected. Otherwise, the stocks with the selected name on all instances associated to the current instance through the selected relationship will be affected by the rate. The *ErrorsTransmit* rate uses a single relationship to the outcome artifact *Errors* stock, summing the number of errors propagated from the income artifacts to the produced artifact when the activity starts. A similar strategy is used by the *ErrorsCommited* rate to add new errors to the produced artifact during the activity accomplishment.

Finally, the *ExpLevel* process calculates the experience level of the team developing the activity. Again we simplify the model for the purpose of this example, assuming that the most experienced developer drives the whole team. So, team experience is equal to the most

experienced developer experience. The *ExpLevel* process uses the GROUPMAX operator to determine the maximum value for a property in the instances associated with the current instance through a selected relationship. Its counterpart, the GROUPMIN operator determines the minimum value for a property in the instances associated with the current instance through a selected relationship.

The above example shows the main components and the syntax of a domain model. Next section presents the model instantiation process, explaining how a model can be built by reusing the knowledge expressed in the domain model.

3.2 Model Instantiation

To exemplify the model instantiation process, consider the software development project presented in Figure 1, where two developers design and code a software module from its specification. Table 6 presents a simplified model for this particular project.

```
DEFINE MyProject ProjectModel
{
        John = NEW Developer
                SET Experience = 1;

        Jimmy = NEW Developer
                SET Experience = 0.8;

        AnalysisModel = NEW Artifact
                SET latent_errors = 10;

        DesignModel = NEW Artifact
                SET latent_errors = 0;

        SourceCode = NEW Artifact
                SET latent_errors = 0;

        Designing = NEW Activity
                SET duration = 10;

        Coding = NEW Activity
                SET duration = 5;
};
```

Table 6 – A simplified model for the proposed project

The model in the preceding table is very simple: it only conveys the instances of the classes defined in the domain model that are necessary to describe the proposed project. The DEFINE keyword introduces the project model, followed by the model name, *MyProject*, and by the domain model to which it is related, *ProjectModel*. Class instances are represented within the model context, delimited by angled brackets.

The developers, *John* and *Jimmy*, are the first class instances presented within the model. The NEW keyword creates an instance of the class identified by the name presented after the keyword. The newly created instance is associated to the identifier presented in the left side of the equal operator. Next, the model creates the instances of the artifacts and the activities.

The SET keyword specifies the value of a property for a specific class instance. Property values are defined immediately after the instance creation. Observe that different property values can be assigned to distinct instances of the same class. For instance, the experience of *John* is supposed to be 1, while the experience of *Jimmy* is supposed to be 0.8. This feature allows system dynamics model developers to precisely account for the relevant differences between instances of a same class, which is harder in the traditional, equation based models.

The preceding project model does not show any relationships between class instances. Table 7 presents a complete model for the proposed project, containing the occurrences of the *Precedence*, *Income*, *Outcome*, and *Team* relationships. The complete model for the proposed project presents class instances, properties, and relationships among class instances. Only the activities specify relationships, since they are always referenced as source classes. The LINK keyword determines which class instances are associated in each relationship. For instance, the *Coding* activity uses the design model artifact as its income, is dependent of the *Designing* activity, is developed by *Jimmy*, and produces the *SourceCode* artifact.

```
DEFINE MyProject ProjectModel
{
	John = NEW Developer
		SET Experience = 1;
	Jimmy = NEW Developer
		SET Experience = 0.8;
	AnalysisModel = NEW Artifact
		SET latent_errors = 10;
	DesignModel = NEW Artifact
		SET latent_errors = 0;
	SourceCode = NEW Artifact
		SET latent_errors = 0;
	Designing = NEW Activity
		SET duration = 10;
		LINK Team John;
		LINK Income AnalysisModel;
		LINK Outcome DesignModel;
	Coding = NEW Activity
		SET duration = 5;
		LINK Team Jimmy;
		LINK Precedence Designing;
		LINK Income DesignModel;
		LINK Outcome SourceCode;
};
```

Table 7 – Complete model for the proposed project

This section presented how models are created from a domain model. Observe that the models do not specify any behavior or system dynamics constructor. Those are inherited from the domain model and used to translate the model description to system dynamics equations that can be simulated. This translation process is called **model compilation**, and is described in the next section.

3.3 Model Compilation to System Dynamics Constructors

The techniques presented in the previous sections allow a model developer to build domain models and create models for particular problems from them. While these techniques help the construction of larger and detailed models, they are rendered useless if these models cannot be simulated. To allow such simulation capabilities, this section describes the process that translates the class-based representation to system dynamics constructors, which can be analyzed in a conventional simulator. This process is called **model compilation**. We will use the software project model presented in Section 3.2 as an example. Table 8 presents the compiled version of this model.

The compiled model conveys only system dynamics constructors, which are represented using the ILLIUM tool modeling language (Barros et al., 2000) (Barros, 2001). This language allows the definition of stocks, rates, processes, and tables. Every constructor has a unique name, used to identify it in the model equations. Processes, stocks, and rates are described by an expression, which is evaluated in every simulation step. Rates are also associated to two stocks, which represent the origin and the target of its flow. Infinity providers, represented by the SOURCE keyword, or infinity sinkers, represented by the SINK keyword, can replace such stocks. Tables are described by a list of comma separated constant values. Tables and infinity sinkers were not necessary in the current example.

```
# Code for object "John"
PROC  John_experience 1.0000;
PROC  John_ExperienceLevel John_experience;

# Code for object "Jimmy"
PROC  Jimmy_experience 0.8000;
PROC  Jimmy_ExperienceLevel Jimmy_experience;

# Code for object "AnalysisModel"
PROC  AnalysisModel_latent_errors 10.0000;
STOCK AnalysisModel_Errors AnalysisModel_latent_errors;

# Code for object "DesignModel"
PROC  DesignModel_latent_errors 0.0000;
STOCK DesignModel_Errors DesignModel_latent_errors;

# Code for object "SourceCode"
PROC  SourceCode_latent_errors 0.0000;
STOCK SourceCode_Errors SourceCode_latent_errors;

# Code for object "Designing"
PROC  Designing_duration 10.0000;
STOCK Designing_TimeToConclude Designing_duration;
RATE  (SOURCE, Designing_TimeToConclude) Designing_Work IF (Designing_DependOk,  -MIN (Designing_ExpLevel
        * Designing_TimeToConclude / DT, Designing_ExpLevel), 0.000);
PROC  Designing_DependOk 0 < 0.001;
STOCK Designing_ExecutingOrDone 0.000;
RATE  (SOURCE, Designing_ExecutingOrDone) Designing_RTExecuting IF (AND (Designing_ExecutingOrDone < 0.001,
        Designing_DependOk), 1.000, 0.000);
RATE  (SOURCE, DesignModel_Errors) Designing_ErrorsTransmit1 IF (Designing_RTExecuting > 0.001,
        (AnalysisModel_Errors) / DT, 0.000);
PROC  Designing_ExpLevel MAX (John_ExperienceLevel);
PROC  Designing_ErrorsPerDay 5.000;
RATE  (SOURCE, DesignModel_Errors) Designing_ErrorCommited1  -1.000 * Designing_ErrorsPerDay *
        Designing_Work / Designing_ExpLevel;

# Code for object "Coding"
PROC  Coding_duration 5.0000;
STOCK Coding_TimeToConclude Coding_duration;
RATE  (SOURCE, Coding_TimeToConclude) Coding_Work IF (Coding_DependOk,  -MIN (Coding_ExpLevel *
        Coding_TimeToConclude / DT, Coding_ExpLevel), 0.000);
PROC  Coding_DependOk (Designing_TimeToConclude) < 0.001;
STOCK Coding_ExecutingOrDone 0.000;
RATE  (SOURCE, Coding_ExecutingOrDone) Coding_RTExecuting IF (AND (Coding_ExecutingOrDone < 0.001,
        Coding_DependOk), 1.000, 0.000);
RATE  (SOURCE, SourceCode_Errors) Coding_ErrorsTransmit1 IF (Coding_RTExecuting > 0.001, (DesignModel_Errors) /
        DT, 0.000);
PROC  Coding_ExpLevel MAX (Jimmy_ExperienceLevel);
PROC  Coding_ErrorsPerDay 5.000;
RATE  (SOURCE, SourceCode_Errors) Coding_ErrorCommited1  -1.000 * Coding_ErrorsPerDay * Coding_Work /
        Coding_ExpLevel;
```

Table 8 – Traditional system dynamics model generated from the model presented in section 2.2

To avoid confusion we will reference the class-based representation by model, while the system dynamics constructors based version will be referenced as the compiled model. The compiled model presents seven distinct blocks, one for each instance represented in the model. Consider the equations generated to the *John* instance of the *Developer* class. Such equations, highlighted in Table 9, convey the declaration of a property and a behavior description.

```
PROC  John_experience 1.0000;
PROC  John_ExperienceLevel John_experience;
```

Table 9 – Model generated to developer John class instance

The first equation declares the *Experience* property for the *John* instance. Properties are declared as processes in the compiled model, being initialized with the value specified to them in the model or by their default value, as stated in the domain model. Observe that the name of the process representing the property in the compiled model is composed by the instance name followed by the property name. Both names, separated by an underlining sign, compose a unique name for the process within the compiled model. This allows the declaration of several instances with their distinct property values, since each instance is required to have a unique name in the model. Different processes in the compiled model will represent such instances. This effect also happens in the model generated to the *Jimmy* instance.

The second equation represents the single behavior description defined in the *Developer* class, which is specialized for the *John* instance. References to properties in the behavior equations are linked to the processes that represent such properties in the current instance. The instance name is also used as a prefix to the behavior constructor name in the compiled model. Behavior descriptions are repeated for every instance in the compiled model. This can also be observed by analyzing the model generated to the *Jimmy* instance.

The model generated for the *Artifact* class instances does not present any particularities that were not highlighted by the *Developer* class instances. However, the model generated for the *Activities* class instances is more interesting. Table 10 highlights the model generated to the *Coding* activity.

```
PROC  Coding_duration 5.0000;

STOCK Coding_TimeToConclude Coding_duration;

RATE  (SOURCE, Coding_TimeToConclude) Coding_Work IF (Coding_DependOk,  -MIN (Coding_ExpLevel *
      Coding_TimeToConclude / DT, Coding_ExpLevel), 0.000);

PROC  Coding_DependOk (Designing_TimeToConclude) < 0.001;

STOCK Coding_ExecutingOrDone 0.000;

RATE  (SOURCE, Coding_ExecutingOrDone) Coding_RTExecuting IF (AND (Coding_ExecutingOrDone < 0.001,
      Coding_DependOk), 1.000, 0.000);

RATE  (SOURCE, SourceCode_Errors) Coding_ErrorsTransmit1 IF (Coding_RTExecuting > 0.001, (DesignModel_Errors) /
      DT, 0.000);

PROC  Coding_ExpLevel MAX (Jimmy_ExperienceLevel);

PROC  Coding_ErrorsPerDay 5.000;

RATE  (SOURCE, SourceCode_Errors) Coding_ErrorCommited1  -1.000 * Coding_ErrorsPerDay * Coding_Work /
      Coding_ExpLevel;
```

Table 10 - Model generated to *Coding* activity class instance

The first equation of the model represents the *Duration* property declaration. The next two equations present behavior descriptions parameterized by other behavior equations and by the *Duration* property. The fourth equation presents the compiled model for the GROUPSUM

operator. Such operator is compiled to a list of arithmetic sums, whose operands are the instances participating in the relationship selected for the operator. The *DependOk* behavior description within the *Activity* class uses the *Precedence* relationship.

In the *Coding* instance, which is preceded by a single activity, the *Designing* instance, the GROUPSUM operator is compiled to a reference to a behavior equation of the *Designing* instance. In the *Designing* activity, the GROUPSUM operator is compiled to zero, since there is no precedent activity, therefore, no operand for the arithmetic sums.

The *DependOk* behavior within the *Coding* instance uses a stock declared by other instance (in this case, the *Designing* activity). This is accomplished through model relationships, which allow an instance to consult or modify the behavior of other instances. The compiling process perceives such access to externally defined behavior through the relationship name, used within the behavior description equations. It precedes the name of the accessed behavior by the name of the instance being manipulated.

The next two equations just replicate behavior descriptions for the current instance. However, the declaration of the *Coding_ErrorsTransmit* rate conveys the model generated to the *ErrorsTransmit* rate declared in the *Activity* class. Such rate is a multirate and shall be compiled to several rate equations, one for each stock that it affects. Since the *ErrorsTransmit* rate is associated to a single relationship in the current example, the model generated for it conveys only one rate, which affects the stock that represents the expected number of errors in the produced artifact (in case, the *SourceCode*). Observe the number appended to the rate name: it is used to differentiate the names of the several rates that can be generated to represent a multirate. Observe also the access to the *Errors* stock in the *DesignModel* instance, which results from the model generated for a GROUPSUM operator.

The next equation uses the GROUPMAX operator, which works in a similar way to the GROUPSUM operator. Instead of generating arithmetic sums for its operands, the GROUPMAX operator applies the MAX operator to its operands. The last two equations present the specialization of a simple behavior description for the instance and the model generated to other multirate.

Although the model generated by the compiling process uses the ILLIUM modeling language, it is based on basic system dynamics constructor. So, the same techniques can be used to generate models for other system dynamics simulators.

4 Related Works

The proposed system dynamics metamodel can be compared to the PowerSim object-oriented system dynamics modeling extensions (Myrtveit, 2000). Both are recent attempts to bring system dynamics modeling to a higher-level of abstraction, where a developer creates models that can be reused by other developers or assembled to build more complex models. However, there are some noticeable differences between both approaches.

First, we believe that the proposed metamodel is simpler than PowerSim object-oriented system dynamics extensions because it comprises a few new concepts and uses plain system dynamics constructors to describe class behavior. PowerSim proposal involves several new elements, such as plugs and sockets, and complex visibility constructors.

Next, our metamodel attempts to describe a problem domain, focusing on the classes of elements that interact within the domain, their relationships, properties, and behavior. A domain expert can provide such high-level modeling, while less experienced developers can build models from the expert-created knowledge based. PowerSim attempts to describe domain-independent components, which can be assembled together by connecting their interfaces to system dynamics constructors from outside or from other components.

Finally, while it clearly defines high-level components to build models with them, PowerSim proposal does not explicitly represent the relationships among such components in

the models. Such relationships have to be built using system dynamics constructors, mixing higher and lower abstraction levels. By focusing in a problem domain and not only on independent components, the proposed metamodel defines the permitted relationships among components (i.e., classes). Class roles are clearly stated and relationships can be used to refine class behavior.

5 Final Considerations and Future Perspectives

This paper presented a system dynamics metamodel, a modeling language that allows the development of domain models and their specialization to particular problems. A domain model describes the classes of elements that compose the domain, detailing their properties, behavior, and relationships. The model represents a particular problem, describing how many elements of each class participate in the problem and defining their property values and associated elements. The model is built using a language closer to the domain concepts, while the metamodel is described using an extended system dynamics syntax. We believe that, since the model construction language is closer to the user knowledge, it helps model development. We have described the domain model construction process, the model instantiation process, and the model compilation process to system dynamics basic constructors. Also, an example of the proposed techniques was presented.

The metamodel compiler to system dynamics was implemented and some domain models were built. We expect to develop more domain models and to create more instances of such models to provide more evidence of the metamodel applicability. We also expect to run some experiments to support our hypothesis that the metamodel produces models easier to develop and understand.

As future perspectives of this work, we are exploring the development of extension modules to domain models. An extension module is a separately described system dynamics model that can be integrated into a model, modifying its behavior without any new system dynamics equation. Such modules are expected to form a scenario library for the domain, being useful for the model developers.

Scenario models and their integration interface to a domain model are relevant to the project within which the system dynamics metamodel was developed. Our intention is to be able to use system dynamics modeling and quantitative risk management through scenario analysis for operational project management. Further details regarding this research can be found in (Barros et al., 2001).

Acknowledgements

The authors would like to thank CNPq, CAPES and FINEP for their financial investment in this work.

References

Abdel-Hamid, T., Madnick, S.E. (1991). *Software Project Dynamics: an Integrated Approach*, Prentice-Hall Software Series, Englewood Cliffs, New Jersey

Barros, M.O., Werner, C.M.L., Travassos, G.H., (2000). "Using Process Modeling and Dynamic Simulation to Support Software Process Quality Management", *XIV Brazilian Symposium on Software Engineering - Software Quality Workshop*, João Pessoa, Brazil

Barros, M.O., Werner, C.M.L., Travassos, G.H. (2001). "Towards a Scenario Based Project Management Paradigm". *Computer Science and Systems Engineering Technical Report 543/01*, COPPE/UFRJ (February)

Barros, M.O. (2001) *ILLIUM - System Dynamics Simulator*, ILLIUM tool homepage at URL http://www.cos.ufrj.br/~marcio/Illium.html

Braga, R., Werner, C., Mattoso, M. (1999) "Odyssey: A Reuse Environment Based on Domain Models", *2nd IEEE Symposium on Application-Specific Systems and Software Engineering Technology (ASSET'99)*, Richardson, USA

de Almeida, F.R., de Menezes S.C., da Rocha A.R.C. (1998) "Using ontologies to improve knowledge integration in software engineering environments", In: *Proceedings of 2nd World Multiconference on Systemics, Cybernetics and Informatics, Volume I / Proceedings of 4th International Conference on Information Analysis and Synthesis*, International Institute of Informatics and Systemics: Caracus, Venezuela; pp. 296–304

Myrtveit, M. (2000) "Object Oriented Extensions To System Dynamics", IN: *The Proceedings of the 18th International Conference of the System Dynamics Society*, Bergen, Norway

Neighbors, J. (1981). "Software Construction Using Components", *Ph.D. Thesis,* University of California, Irvine, USA

Oliveira, K.M., Rocha, A.R.C. Travassos, G.H., Menezes, C. (1999). 'Using Domain-Knowledge in Software Development Environments", *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*, Kaiserslautern, GR (June)

Pressman, R.S. (1997). *Software Engineering: a Practitioner's Approach*, Fourth Edition, McGraw-Hill Companies Inc.

---

[i] We shall inforce that this is a simplified view of the proposed domain, just for the sake of the example. A formal model can de found in (de Almeida et al., 1998).

[ii] BNF stands for *"Backus Naur Form"*, which is a formal notation to describe the syntax of a programming language.