Enhancing Metamodels with Scenarios: Plug-&-Simulate Extensions for Model Developers

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 2562-8675
{marcio, werner, ght}@cos.ufrj.br

Abstract

In a previous work, we presented the concept of metamodels, an extension to system dynamics that allows the development and specialization of domain models. A domain model provides a high-level representation for the elements that compose a knowledge area. Specific models developed for the domain are based on these elements, inheriting their behavior from the domain model. Traditional system dynamics constructors (stocks, rates, and processes) describe the behavior of domain elements. Domain models are believed to simplify model development within a knowledge area.

In this paper, we present scenario models, which act as "plug-&-simulate" extensions to domain models. A scenario model allows a developer to change the behavior of domain elements without direct and error-prone intervention in the domain model equations. While analyzing a model, a developer can select relevant scenarios and activate them upon the model components. The model behavior is adjusted for the selected scenarios, presenting their impact upon the original model behavior. We present the system dynamics metamodel, the structure of scenario models, and their integration with specific models.

KEYWORDS: system dynamics metamodel, model extensions

1 Motivation

In a previous work, we presented the concept of metamodels for system dynamics model development (Barros et al., 2001a). By using metamodels, we propose a different approach for modeling, where specific models for a knowledge area are developed based on a domain model. The domain model conveys specifications for relevant elements of the knowledge area, describing their behavior in terms of system dynamics traditional constructors – stocks, rates, and processes. Specific models for the knowledge area contain no system dynamics constructors, but instances of the domain elements, specialized for the model under interest.

Domain models attempt to group the equations that describe the behavior of each relevant element within a domain, in order to improve model readability and understanding. In a traditional model development effort, the relevant elements that compose a knowledge area are not easily identified in a maze of system dynamics constructors. Their representation is usually spread among several equations, which forces developers to analyze the whole model to determine the precise group of equations that describe the behavior of an element and its relationships to other elements.

Domain models also allow the reuse of domain knowledge when building models. In a traditional modeling effort, model equations convey knowledge for the problem under interest, but they also contain knowledge that could be shared by several models within the same domain. This domain knowledge should be repeated in every particular model developed for the domain. By using the system dynamics metamodel, domain knowledge is separately represented in a domain model, while every model developed for the domain inherits and specializes this knowledge.

Scenario models are extensions to domain models that address the issue of separating facts from assumptions within a model. Traditionally, system dynamics models blend known facts about the real-world elements that compose a model with several assumptions upon their behavior and interaction. Most of these assumptions are controlled by parameters that allow them to be activated or deactivated during a specific simulation. By running different simulations, injecting distinct configurations of values into these parameters, an analyst can evaluate the implications of the assumptions upon the model behavior. However, it is usually difficult to test other assumptions than those provided with the model: model equations should be changed to incorporate the behavior that is sought. Changing model equations is a error-prone process, specially in large models with hundreds or thousands of equations.

By separating facts from assumptions, a model analyst can build a baseline model, containing only known facts about a knowledge area, and separate models describing the uncertain assumptions. The model analyst simulates the baseline model to evaluate its behavior without any assumptions. Next, he performs an iterative process, where assumption models are integrated to the baseline model and the combined model is simulated to evaluate how the assumptions affect the baseline model behavior. In the metamodel framework, a specific model for a domain represents known facts, while scenario models separately describe assumptions.

This paper is organized in five sections. The first one comprises this motivation. Section 2 briefly describes the system dynamics metamodel, presenting an example that will be further used to illustrate scenario models. Section 3 shows the structure of scenario models and their integration to a baseline model. Section 4 presents an application example of the system dynamics metamodel and scenario models. Finally, section 5 presents some final considerations and future perspectives of this work.

2 The System Dynamics Metamodel

The purpose of this paper is to present scenario models and how they can be used to evaluate the impact of assumptions upon a system dynamics model. However, due to scenarios straight link to the system dynamics metamodel, we briefly describe the concepts that compose the metamodel, as presented in (Barros et al., 2001a). Also, we introduce an example, based on the classic bathtub model, to illustrate the metamodel concepts. Further, this example will be used to describe scenario models.

The system dynamics metamodel proposes a three-staged process for model development. First, an expert in a given knowledge area builds a domain model, conveying descriptions for the relevant elements that compose a domain. Each element is described as a *class*, which contains the *properties* that describe the element and its behavior equations. Traditional system dynamics constructors describe a class behavior. The domain model also contains declarations for acceptable *relationships* among classes. A relationship is a directed connection between two classes that allows behavior equations for one class (relationship source) to access information and behavior equations in the other class (relationship target).

Table 1 presents a simple domain model describing a bathtub with two kinds of valves: sources, which add water to the bathtub, and sinkers, that drain water from the bathtub. The behavior that describes the whole system of bathtub and valves is separated in three classes.

The bathtub behavior, presented in the *Bathtub* class, is stated by a single stock that describes the amount of water within the bathtub. Behavior equations for sources and sinkers, which are presented in their respective classes, declare rates that affect the level of the bathtub stock. These rates use class relationships to address the stock. Also, sources and sinkers have properties, describing the amount of water that they allow in or out of the bathtub.

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 (Barros et al., 2001a). Observe that the domain model in Table 1 does not specify how many sources or sinkers are connected to a bathtub. It only states that sources and sinkers can be connected to a single bathtub (class relationships) and how they behave when this connection is established. The relationships defined in the domain model also prevent incorrect class connections. For instance, the domain model in Table 1 does not allow a source to be directly connected to a sinker.

Table 1 – A domain model for the bathtub example

```
MODEL BathtubModel
{
        CLASS Bathtub
        {
                STOCK Level 0;
        };

        CLASS Source
        {
                PROPERTY GrowthRate 10;

                RATE (SourceTub.Level) Growth GrowthRate;
        };

        CLASS Sinker
        {
                PROPERTY SinkRate 10;

                RATE (SinkerTub.Level) Sink -Min(SinkRate, SinkerTub.Level);
        };

        RELATION SourceTub Source, Bathtub;
        RELATION SinkerTub Sinker, Bathtub;
};
```

Domain model classes are used as high-level constructors for models developed within the domain. The group of classes described in a domain model composes a domain specific modeling language. Several models can be built from the same domain model. Model developers interested in describing a problem within that knowledge area specify the problem in terms of classes, reusing their behavior equations from the domain model. To avoid confusion with domain models, we will refer to models developed for a domain as *specific models for the domain* or just *specific models*.

In the second stage of the model development process, a developer specifies how many *instances* of each class defined for the domain exist in the specific model of interest. For instance, a model developed for the "bathtub domain" specifies how many sources and sinkers are connected to each bathtub, along with the amount of water that they drain or put into the bathtub per simulation step. Table 2 presents a model developed from the bathtub domain model, containing a single bathtub, one sinker, and one source.

Each model element (bathtub, sinker, and source) is defined as an instance of a domain model class. The property values for the instances are specified after their declaration. In the model presented in Table 2, the source does not specify its *GrowthRate* property value. So, this property assumes its default value (10 ml) for the instance. In contrast, the sinker defines its *SinkRate* property value, indicating that it drains 5ml of water per simulation step.

Table 2 – Specific model for the bathtub domain

```
DEFINE MyBathtub BathtubModel
{
        Bathtub = NEW Bathtub

        Sourcer1 = NEW Source
                LINK SourceTub BathTub;

        Sinker1 = NEW Sinker
                SET SinkRate = 5;
                LINK SinkerTub BathTub;
};
```

The model also describes how instances relate to each other, based on the relationships among classes defined in the domain model. In the model presented in Table 2, the source and the sinker are associated to the *Bathtub* instance. If several bathtubs, sinkers, and sources composed the whole system, the class relationships would allow each sinker or source to indicate the bathtub that it affects. So, the rates within these elements would address the stock that describes the level of the correct bathtub.

A specific model for a domain conveys only information about the elements that it uses. It does not present any system dynamics constructor. Such constructors are inherited from the classes' behavior, which are described in the domain model. By using the system dynamics metamodel, we expect that modeling becomes easier than using pure system dynamics constructors, since model developers use domain elements described by the domain specific language to build their models.

Regarding the property values in the bathtub example, every class instance has different property values. 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, which are difficult to handle manually. By using the metamodel, the model developer just defines the individual property values for the instances and the model behavior automatically adjusts to these values. Behavior equations can be parameterized by the property values of each individual instance, generating different behavior for elements with distinct characteristics.

Finally, in the third step of the model development process, the specific model is translated to system dynamics equations in order to be analyzed in standard system dynamics simulators. The resulting model uses only traditional constructors, while the preceding model is described in the high level, element-oriented 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. Table 3 presents the compiled version for the model presented in Table 2. To reduce the number of equations in the compiled model, reducing thus the time required for its simulation, compiler level optimizers can further process the model.

Table 3 – Compiled model (without optimization) for the model presented in Table 2

```
# Code for object "Bathtub"
STOCK Bathtub_Level 0;

# Code for object "Sourcer1"
PROC Sourcer1_GrowthRate 10;
RATE (SOURCE, BathTub_Level) Sourcer1_Growth1 Sourcer1_GrowthRate;

# Code for object "Sinker1"
PROC Sinker1_SinkRate 5;
RATE (SOURCE, BathTub_Level) Sinker1_Sink1 -MIN (Sinker1_SinkRate, BathTub_Level);
```

3 Scenario Models

Scenario models address the problem of analyzing hypothesis upon a system dynamics model. Usually, a model has several parameters that control the activation of predefined assumptions, supporting the analysis of a model under distinct conditions. However, when a different assumption, theory, strategy or uncertain event has to be analyzed upon a model, model equations have to be changed to include the new behavior before it could be simulated. Even more, the recently introduced equations can contradict some of the original behavior, eventually resulting in inconsistencies within the model. Finally, changes within a model can introduce errors, affecting both the original and the innovative behavior.

Scenario models are extensions to domain models that propose the separation of facts from assumptions. A scenario is developed for a particular domain and provides new behavior and characterization for one or more domain classes. From the separation point of view, a specific model for a domain contains known facts, while scenarios for the same domain represent uncertain assumptions.

A scenario model is composed by connections and constraints. A *connection* associates the scenario to a domain class, so that the scenario can be enacted upon instances of the class in a specific model for the domain. A *constraint* declares restrictions to which the connected instances and its associated instances have to apply in order to use the scenario. Table 4 presents a simple scenario model for the bathtub domain.

Table 4 – A simple scenario model for the bathtub domain

```
SCENARIO SplashSource BathtubModel
{
        CONNECTION TheSource Source
        {
                PROPERTY SplashPeriod 5;

                PROC Div TIME / SplashPeriod;
                PROC RDiv Round (TIME / SplashPeriod);
                PROC TimeToSplash AND(Div-RDiv < 0.001, Div-RDiv > -0.001);

                AFFECT Growth if(TimeToSplash, Growth, 0);
        };
};
```

The scenario in Table 4 models a “splashing source”, that is, a non-continuous source that allows water into the bathtub in periodic turns. The scenario has a single connection and no constraints. The *TheSource* connection allows the scenario to be connected to an instance of the *Source* class in specific models developed for the *Bathtub* domain.

A connection extends the behavior and characterization of its associated class by adding new properties and behavior equations. It also declares behavior redefinition clauses, which allow the scenario to change behavior equations previously defined for the class in the domain model. The *TheSource* connection in the scenario model presented in Table 4 declares a single property, namely *SplashPeriod*, which indicates the water-supplying period for the splashing source. The connection also presents three auxiliary behavior equations, represented by processes (PROC) in the scenario model, which are ultimately used by a behavior redefinition clause.

Scenarios are developed to adjust the equations of rates and processes defined in the domain model. In the scenario in Table 4, the single behavior redefinition clause, represented by the *AFFECT* keyword, indicates that the scenario modifies the *Growth* equation defined for the *Source* class in the domain model. The original equation that describes a source’s water-supplying rate is overridden by the scenario definition. In this scenario, the new *Growth* equation refers to the original equation. However, it uses the *TimeToSplash* process, which signals the simulation steps when the splashing source shall pour water into the bathtub, to turn the source intensity to zero out of the periodic supply turns.

Like the domain model, a scenario is not a self-contained model. It is a complementary model that adjusts the behavior of previously developed models. A scenario can be activated upon a model developed for the same domain to which the scenario was created. When a scenario is activated upon a model, its connections must be enacted upon class instances declared within the model. The effects of enacting a connection upon an instance are similar to declaring the properties and behavior equations defined in the connection directly in the domain model class. However, if such properties and behavior equations were declared in the domain model, they would apply for every instance of the class in every specific model developed for the domain. Scenario connections can be enacted upon specific class instances, modifying the behavior of those particular instances. Table 5 shows a *SplashSource* scenario activation upon the *Sourcer1* instance in the specific model presented in Table 2.

Table 5 – Scenario model activation upon a model for the bathtub domain

```
DEFINE MyBathtub BathtubModel
{
        Bathtub = NEW Bathtub

        Sourcer1 = NEW Source
                LINK SourceTub BathTub;

        Sinker1 = NEW Sinker
                LINK SinkerTub BathTub;

        ACTIVATE SplashSource
                CONNECT TheSource Sourcer1;
};
```

Since no splashing source was defined in the domain model, a specific model for the domain cannot directly contain such kind of source. Instead, the model shall create an instance of a conventional, continuous source and activate the *SplashingSource* scenario upon this source. Table 6 presents the compiled version for the model in Table 5.

Table 6 – Compiled version for the model in Table 5 (accounting for scenario equations)

```
# Code for object "Bathtub"
STOCK Bathtub_Level 0;

# Code for object "Sourcer1"
PROC Sourcer1_GrowthRate 10;
PROC Sourcer1_SplashPeriod 5;
RATE (SOURCE, BathTub_Level) Sourcer1_Growth1 IF(Sourcer1_TimeToSplash,
     (Sourcer1_GrowthRate), 0);
PROC Sourcer1_Div TIME / Sourcer1_SplashPeriod;
PROC Sourcer1_RDiv ROUND (TIME / Sourcer1_SplashPeriod);
PROC Sourcer1_TimeToSplash AND (Sourcer1_Div - Sourcer1_RDiv < 0.001, Sourcer1_Div -
     Sourcer1_RDiv >  -0.001);

# Code for object "Sinker1"
PROC Sinker1_SinkRate 5;
RATE (SOURCE, BathTub_Level) Sinker1_Sink1  -MIN (Sinker1_SinkRate, BathTub_Level);
```

By comparing the compiled model presented in Table 6 with its original version, generated without the scenario activation and presented in Table 3, we observe that the equations for the *Bathtub* and *Sinker1* instances were not modified. This occurs because no scenario connection was enacted upon those instances. However, the equations defined in the *TheSource* connection of the *SplashingSource* scenario were added to the *Sourcer1* instance description. The *Div*, *RDiv* and *TimeToSplash* processes were declared for the instance, while its *Growth* rate was redefined as indicated in the behavior redefinition clause in the scenario.

The properties defined by a scenario connection are added to the list of properties that describe the class instance upon which the connection was enacted. As it occurs in the domain model, these properties have a default value, which can be redefined by particular instances in a specific model. Table 7 presents an extract for the model in Table 5 where the *Sourcer1* instance redefines the value of its *SplashPeriod* property. The connection equations assume the new property value for the instance, adjusting scenario behavior for this value. Observe that, if no scenario connections were enacted upon the *Sourcer1* instance, the initialization of the *SplashPeriod* property would result in an error, since the property was not defined for the class in the domain model.

Table 7 – Scenario model activation upon a model for the bathtub domain

```
DEFINE MyBathtub BathtubModel
{
        ...

        Sourcer1 = NEW Source
                SET SplashPeriod = 10;
                LINK SourceTub BathTub;
        ...

        ACTIVATE SplashSource
                CONNECT TheSource Sourcer1;
};
```

The *SplashSource* scenario and the *Bathtub* domain model help to illustrate the main advantage of using scenario models. Consider that a model analyst wants to evaluate the bathtub water level behavior with a continuous source and with a splashing source. To evaluate the first, the analyst uses the specific model for the *Bathtub* domain presented in Table 2. This model shows the behavior presented in the left-hand graph in Figure 1. Next, to evaluate the splashing source behavior, the analyst activates the *SplashingSource* scenario upon the continuous source, generating the model presented in Table 5. This model shows the behavior presented in the right-hand graph in Figure 1.

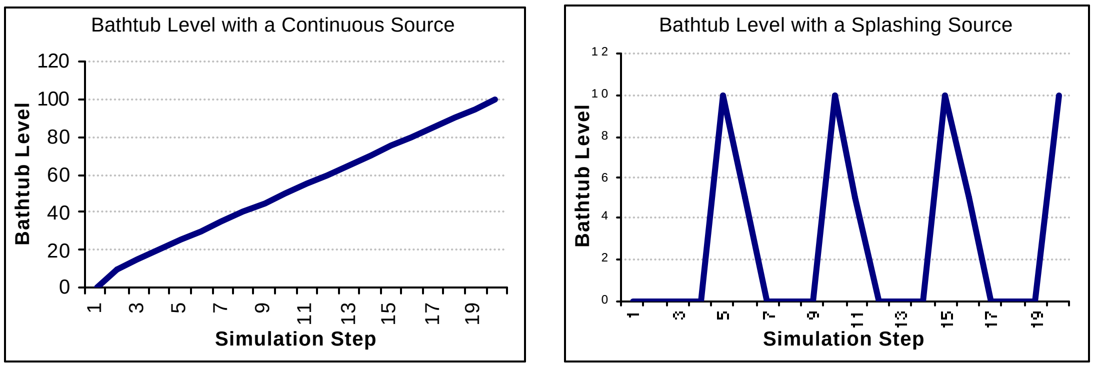

Figure 1 – Model behavior with and without scenario activation

So, scenarios allow an analyst to perform behavior analyses upon a model without direct intervention in its equations or even parameter redefinitions. Scenarios are separately modeled and activated upon model instances. They act as “plug-&-simulate” extensions to a domain model: they provide different behavior for domain model classes that can be plugged and analyzed, according to the analyst needs. These analyses can be rather difficult with the current textual representation for scenarios and models, since users need to change the model representation, but graphical tools for specific model development, scenario creation and integration can be built to allow a user to graphically build a model and activate scenarios upon it.

Scenarios are also reusable along many models developed for a domain. They separate the knowledge about an uncertain event, assumption, theory, or strategy from the general, and usually more reliable, domain knowledge. The integration of scenarios to a specific model for a domain may reveal changes in the specific model behavior. Simulating a model with a scenario demonstrates the potential impact of the scenario in the model behavior. This kind of simulation allows the model analyst to test project sensitivity to the assumptions expressed by the scenario model. Scenario models can only be simulated when integrated to a specific model

3.1 Scenarios with Multiple Connections

Besides the basic fundamentals for scenario development, more complex scenarios can be built. Table 8 presents a scenario model with two connections. This scenario describes a periodic source and a periodic sinker. Since its behavior involves two domain model classes (*Source* and *Sinker*), the scenario must have two connections. Each connection declares independent properties, behavior equations and behavior redefinition clauses for its class.

Table 8 – A scenario model with two connections for the bathtub domain

```
SCENARIO AlternateTubs BathtubModel
{
        CONNECTION TheSource Source
        {
                PROPERTY BaseStart 0;
                PROPERTY Period 2;

                STOCK Clock BaseStart+Period;
                RATE (Clock) ClockRT if(Clock < 0.001, Period, -1);
                AFFECT Growth if (Clock < 0.001, Growth, 0);
        };

        CONNECTION TheSinker Sinker
        {
                PROPERTY BaseStart 1;
                PROPERTY Period 2;

                STOCK Clock BaseStart+Period;
                RATE (Clock) ClockRT if(Clock < 0.001, Period, -1);
                AFFECT Sink if (Clock < 0.001, Sink, 0);
        };
};
```

When a scenario with several connections is activated upon a model, every connection must be enacted upon a class instance. Table 9 presents an extract from a specific model where the *AlternateTubs* scenario is activated. The model presents the enactment of the *TheSource* and *TheSinker* connections upon the *Sourcer1* and *Sinker1* instances, respectively.

Table 9 – An extract of a specific model, focusing on the *AlternateTubs* scenario activation

```
DEFINE MyBathtub BathtubModel
{
        ...

        ACTIVATE AlternateTubs
                CONNECT TheSource Sourcer1;
                CONNECT TheSinker Sinker1;
};
```

3.2 Constrained Scenarios

Scenarios may depend on other scenarios to represent their behavior. Suppose, for instance, a splashing source whose water-supplying intensity is stochastic and uniformly distributed. The scenario presented in Table 10 could represent this effect.

Table 10 – A constrained scenario model for the bathtub domain

```
SCENARIO RandomSplashSizeSource BathtubModel
{
        CONNECTION TheSource Source
        {
                AFFECT Growth if(TimeToSplash, Growth * (0.5 + Uniform(0,1)), 0);
        };

        CONSTRAINT TheSource, SplashSource.TheSource;
};
```

The scenario in Table 10 redefines a source's *Growth* rate, modeling the stochastically driven source. Note that the equation redefinition uses the *TimeToSplash* process that is not defined in the scenario, nor in the domain model. This process is defined in the *SplashSource* scenario, which must be enacted upon the same instance upon which the *TheSource* connection from the *RandomSplashSizeSource* scenario was enacted.

The constraint in the scenario warrants the establishment of this connection. It states that instances affected by the *TheSource* connection in the *RandomSplashSizeSource* scenario must also be affected by the *TheSource* connection of the *SplashSource* scenario. If the last connection is not enacted upon the instance, the metamodel compiler issues an error and does not generate the compiled model.

Constraints are not restricted to class instances upon which a scenario connection is enacted. Other instances, linked to the connected instances by class relationships, can also be affected by constraints. To allow associated instances evaluation by a constraint, a dot operator and a relationship identifier should follow the *TheSource* connection in the left-hand side of the comma that divides the constraint declaration. All class instances associated to the connected instance through this relationship should attend to the scenario connection presented by the right-hand side of the comma.

3.3 Scenario Activation Ordering

Since several scenarios can redefine the same equation for a class instance in a specific model, and due to operator precedence rules within an equation, scenario activation ordering is relevant. If several scenario connections are enacted upon the same class instance, their behavior redefinition clauses affect the domain class equations according to the scenario activation order.

Consider the scenarios presented in Table 11. The first scenario represents a pumped source, where a pumping engine enhances the water-supplying rate by a multiplying factor. The second scenario represents a dripping source, where a hole reduces the water-supplying rate by a constant amount per simulation step. Both scenarios affect a source's *Growth* rate, but the combined effect of enacting their connections upon the same class instance depends on the order that they were activated upon the model.

Consider that the *TheSource* connection in the *PumpedSource* scenario was enacted upon a class instance and then the *TheSource* connection in the *DrippingSource* was enacted upon the same instance. The resulting *Growth* equation, amplified by the pump factor and then reduced by the hole dripping rate, would be described as:

```
Growth = (GrowthRate * PumpRate) – HoleSize
```

Table 11 – Scenarios that affect a source's water supplying rate

```
SCENARIO PumpedSource BathtubModel
{
        CONNECTION TheSource Source
        {
                PROPERTY PumpRate 2;
                AFFECT Growth Growth * PumpRate;
        };
};

SCENARIO DrippingSource BathtubModel
{
        CONNECTION TheSource Source
        {
                PROPERTY HoleSize 1;
                AFFECT Growth Growth - HoleSize;
        };
};
```

However, if scenario activation ordering changed, enacting the *TheSource* connection in the *DrippingSource* scenario before the *TheSource* connection in the *PumpedSource* scenario upon the same instance, the *Growth* equation would be such as:

```
Growth = (GrowthRate – HoleSize) * PumpRate
```

In the second activation order, the hole effects are perceived prior to the pump amplification effects. Depending on property values (hole size and pump rate), these two equations would show distinct behavior in a specific model. So, scenario ordering must be considered when connections from several scenarios are enacted upon the same class instance.

4 An Application Example

The *Bathtub* domain is a simple example, presented in this paper to illustrate the structure of scenario models, how they are integrated to a specific model for a domain, and how this integration affects the model behavior. A more complex application of the system dynamics metamodel and scenarios models can be found in the scenario based project management paradigm (Barros et al., 2001b; Barros et al., 2002).

The scenario based project management is a paradigm for software project management that proposes that a manager should plan and document the expected behavior for a project as a model. Since this behavior can be affected by unexpected events during project development, the manager should test its sensibility to several combinations of such events, getting feedback about possible risks that can challenge the project success. Project management scenarios represent such events, conveying knowledge that can be reused along several projects.

In this context, the expected behavior for a project is modeled as a specific model for the project management knowledge area. A domain model was developed for this knowledge area and several scenarios were built upon it. Currently, we have about twelve scenario models developed for the domain. These scenarios include theories regarding developers' productivity and error generation rates due to their experience, developers' productivity due to learning the application domain, effects of overworking and exhaustion upon software developers, communication overhead, error propagation along the activities that compose a work breakdown structure, among others.

In an industrial setting, where the scenario based project management paradigm can be used to manage real software projects, senior managers develop scenario models expressing the experiences they have collected by participating in several projects. These scenarios allow less experienced managers to share senior managers' knowledge. In an academic setting, scenarios developed by experts and supported by the software engineering literature can be useful to training activities. For instance, based on a proposed software project, trainees could use scenario integration and simulation to evaluate the impact of their decisions upon the project behavior (cost, schedule, quality, and so on).

Scenarios are supposed to be small: they shall concentrate on the behavior equations that describe a particular problem or opportunity. The power of scenarios is their integration with specific models and the dependencies among scenarios that can be described through constraints. For instance, the scenario that describes the effects of exhaustion upon developers' productivity and error generation rates depends on the scenario that describes the effects of overworking upon developers. Since further scenarios can describe other effects of overworking upon developers (beyond productivity and error generation rates), the effects of overworking and exhaustion were separated and described by two related scenarios.

5 Final Considerations and Future Perspectives

This paper presented scenario models, an extension to the system dynamics metamodel that allows the separation of uncertain assumptions from the facts expressed in a model. Such uncertain assumptions are described in separate models, namely scenario models, which can be activated upon model components. Such activation adjusts the original model to the equations that describe the scenario, allowing a model analyst to evaluate the impact of these assumptions upon the model behavior. Scenarios allow model developers to extend the behavior of a domain model without direct and error-prone intervention in its equations.

The metamodel compiler to system dynamics constructors was adjusted to account for scenario activation and some scenario models were built, mostly for the project management knowledge area. We expect to create new scenarios for this domain and use the proposed techniques as a training tool. Software project manager trainees would be given a project description and a set of scenarios to evaluate their impact upon project behavior. Recently, we have conducted an experimental study that showed indications that managers using scenarios to support their decisions perform better than managers that support their decisions only upon personal experience.

Future perspectives of this work include the development of graphical tools to support the creation and evaluation of specific models and scenario models. Such tools would be useful as a simulation environment, where a model analyst could select relevant scenarios for the analysis context and easily activate them upon a model developed for the same domain. We also expect to apply scenarios and the system dynamics metamodel to training by developing a scenario-based training environment.

Acknowledgements

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

References

Barros, M.O., Werner, C.M.L., Travassos, G.H. (2001a), "From Metamodels to Models: Organizing and Reusing Domain Knowledge in System Dynamics Model Development", in Proceedings of the 19th International Conference of the System Dynamics Society, Atlanta (July)

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

Barros, M.O., Werner, C.M.L., Travassos, G.H. (2002), "Project Management Knowledge Reuse Through Scenario Models", accepted for publication in Proceedings of the Seventh International Conference on Software Reuse, Texas (April)

Back to the Top