Proceedings of the 2011 Winter Simulation Conference
S. Jain, R.R. Creasey, J. Himmelspach, K.P. White, and M. Fu, eds.

IMAGE-SCENARIZATION: FROM CONCEPTUAL MODELS TO EXECUTABLE SIMULATION

François Rioux

LTI Software and Engineering
825 boul. Lebourgneuf, Bureau 204,
Québec, QC G2J 0B9 CANADA

Michel Lizotte

Defence R&D Canada – Valcartier
2459, Pie-XI Blvd. North,
Québec, QC G3J 1X5 CANADA

ABSTRACT

Agent-based modeling has proven to be a natural way to express various types of problems or situations. Some research has focused on the analysis and design of agent models, but little has truly addressed the need for automated assistance when creating agent-based models from the initial problem comprehension. This paper proposes an approach addressing this gap and supporting the iterative process of generating executable agent models. In particular, this approach enables the incremental conceptual representation of a problem and the development of agent models. This paper presents how to develop an agent-based model using a predefined generic Scenarization Vocabulary. It then describes the technical approach that was chosen in order to exploit the initial conceptual design and facilitate the development of models by eliminating software development technicalities. This approach is part of a broader research effort known as IMAGE, which proposes a toolset supporting collaborative understanding of complex situations.

1 INTRODUCTION

A simulation based on current beliefs about a situation is either a mental act or an actual execution of one's explicit model using a software tool to better predict the future. For instance, to increase their grasp of a situation, the military use wargaming to imitate the dynamics between different forces. The use of an executable model built from the current comprehension of a situation is a way to facilitate this action and better appreciate future possibilities.

A large community of research is investigating agent-based modeling. Work by Macal and North (2006) discusses the analysis and design of such software. However, no study seems to truly address the need for automated assistance when creating agent-based models from the initial problem comprehension. In a recent paper, Lizotte and Rioux (2010) have introduced the IMAGE Scenarization (IMAGE-SCE) approach supporting a spiral process of generating a simulator and addressing the lack of formalism and procedural maturity stressed by Macal and North (2006). The current paper presents the generic agent-based simulation (GABS) framework part of the IMAGE-SCE approach, which facilitates the development of actual models.

The work is part of the IMAGE concept (Lizotte et al. 2008), which targets the collaboration of experts trying to reach a common understanding of a complex situation. The principles underlying the concept include: (1) Iterative understanding: a common understanding is reached through revision and sharing of successive representations of the situation; (2) Synergy of technologies: the common understanding is assisted by the synergy of tools for representation, scenarization, simulation and exploration; and (3) Humans in the loop: achieving a common understanding is above all a human task, supported by tools helping each person's comprehension process. This paper focuses on the 2nd principle, and in particular details the technical approach of the IMAGE-SCE concept, which consists of building agent-based executable simulation models that are expressed using conceptual graphs of predefined semantics. The pro-

cess used is said to be "spiral" due to its iterative and incremental nature. Section 2 introduces how agents are modeled via conceptual graphs, and introduces the modeling semantics. Section 3 details the GABS framework that is used to execute the models as well as the source code generation principle that facilitates model implementation. Finally, Section 4 presents related work.

2 CONCEPTUAL MODELING OF A SITUATION

2.1 Conceptual Graphs

Figure 1 provides an example of the graphical notation of a conceptual graph formalism (Sowa 1984) variant used in the current approach. This example was produced using CoGUI-IMAGE, which is based on the CoGUI tool (LIRMM 2011). The latter software is founded on the conceptual graph research efforts of Chein and Mugnier (2008) as well as on work by Genest (2010). Rectangles are *concepts*, while ellipses are *conceptual relations*. A concept has two parts: a *type label* before the colon and a *marker* after the colon. The type label represents the type of entity the concept refers to, while the marker refers either to the generic marker "*" or identifies actual individuals, also called *referents*. Pairs of arcs pointing toward or away from an ellipse mark arguments of the relation (all relations are dyadic). The arrow orientation is devoid of semantics and serves only to ease graph reading. A concept can also be detailed using a nested graph, as shown in Figure 1, where the concept "IED Event:*" (IED stands for "Improvised Explosive Device") is detailed with the graph "IED:*→ patient→Explode:*". This part of the graph means that an *IED Event* is actually an IED that explodes. The *IED* concept (a kind of entity) is in relation with the *Explode* concept (a kind of behavior) through the relation *patient* that links a passive entity (a patient; see 2.2, below) to a reaction behavior.

Conceptual Graphs are a system of logic based on the existential graphs of Charles Sanders Peirce (Peirce 1933) and the semantic networks of artificial intelligence (Sowa 1984). They express meaning in a form that is logically precise, humanly readable and computationally tractable. They can be used as an intermediate language for translating computer-oriented formalisms to and from natural languages. They serve as a readable, but formal, design and specification language. They have been implemented in a variety of projects for information retrieval, database design, expert systems and natural language processing (Sowa 1984). This is why they were chosen to represent conceptually the comprehension of a situation.

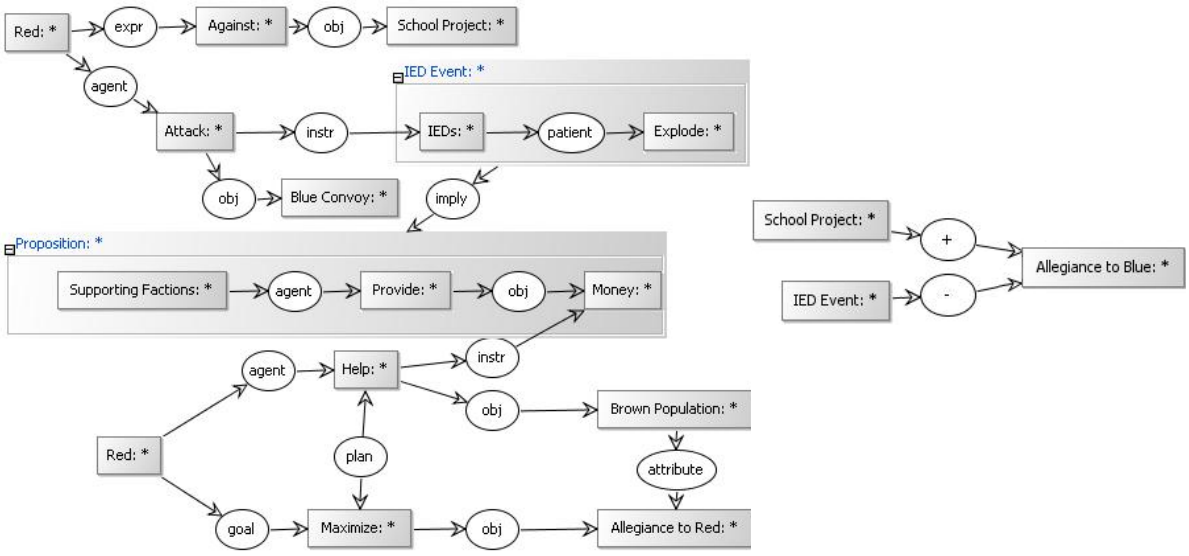

Figure 1: Conceptual graph illustrating a comprehension model

The first step in building the conceptual representation of a situation is to identify concepts and relations between these concepts, without any semantic restrictions but the ones imposed by the conceptual graph notation. This step enables the designer to express the main elements that are part of a situation, including the key ones that will be part of the simulation, and to build a starting point toward an improved comprehension. The resulting graph set provides a solid conceptual base but is not sufficient to automatically create a simulation model skeleton: it is too complicated and error-prone to interpret such free-form conceptual graphs. Therefore a second modeling step is required to evolve the initial comprehension graphs and create new ones to be interpreted and exploited by the software producing the simulation model skeleton.

2.2 Scenarization Semantics

Beginning with the comprehension concepts and relations introduced above, the designer has to identify objects relevant to the simulation. While the Comprehension Model (CM) leaves the designer free to express any knowledge related to the situation, the Scenarization Model (SM) requires that all objects essential to simulate the situation be consistent and defined at the correct level of detail using the IMAGE-SCE semantics. This is accomplished using both the Scenarization Vocabulary and the Specification Schemas (Chein and Mugnier 2008). As for the CM, a set of graphs and the Scenarization Vocabulary constitute the Scenarization Model. The main scenarization concept types are:

- *Agent:* a Simulation Object Entity (or Actor) that voluntarily, under some "Motivation Rules," performs Actions trying to modify the situation, and whose Reactions, under some "Reflex Rules," also modify the situation.
- *Patient:* a Simulation Object Entity (or Actor) that never acts under "Motivation Rules," but whose Reactions, under some "Reflex Rules," modify the situation.
- *Decor:* a Simulation Object Entity part of the environment (not an Actor) that is accessed by the Actors, but not modifiable by them.
- *Action:* a Simulation Object Behavior triggered by an Agent.
- *Reaction:* a Simulation Object Behaviour triggered by an Actor (Agent or Patient).
- *Variable:* a Simulation Object Attribute changed by the simulation execution. Also called a *dynamic property*.
- *Parameter:* a Simulation Object Attribute that is set by the designer and that cannot be changed by the simulation execution. Also called a *static property*.
- *Predicate:* a Simulation Object Proposition constructed with a Simulation Object Entity, a relation, and either an Object Attribute or a Simulation Object Behavior.
- *Cycle:* a simulation construct, composed of *cycle steps*, which specifies either the order in which behaviors should be executed or the priority in the evaluation of stimulations (motivations or reflexes).
- *Scenario:* a simulation construct that contains various global Parameters, all Actors that should take part in the simulation, as well as the definition of Cycles that provides an initial order of execution for the behaviors.

The simulation is single-threaded and event-driven, with clock ticks as one possible event. Although they serve different purposes, the CM and the SM are closely related. Any meaningful simulation object part of the SM implements one or many comprehension concepts. Some comprehension elements will not be part of the SM, either because they are not relevant to the simulation or because they are not mature enough to be integrated at the current stage of the spiral process. Conversely, some elements will be introduced in the SM to add details required for the actual execution of the simulation. In other words, the designer needs to isolate comprehension concepts and relations that are relevant for the simulation, and add concepts and relations required for simulation purposes. Figure 2 shows a vocabulary resulting from this activity. The generic Scenarization Vocabulary (e.g., entity, actor, s-param) builds on previous work on planning systems such as STRIPS in Nilsson (1980) and SAIRVO in Lizotte (1989).

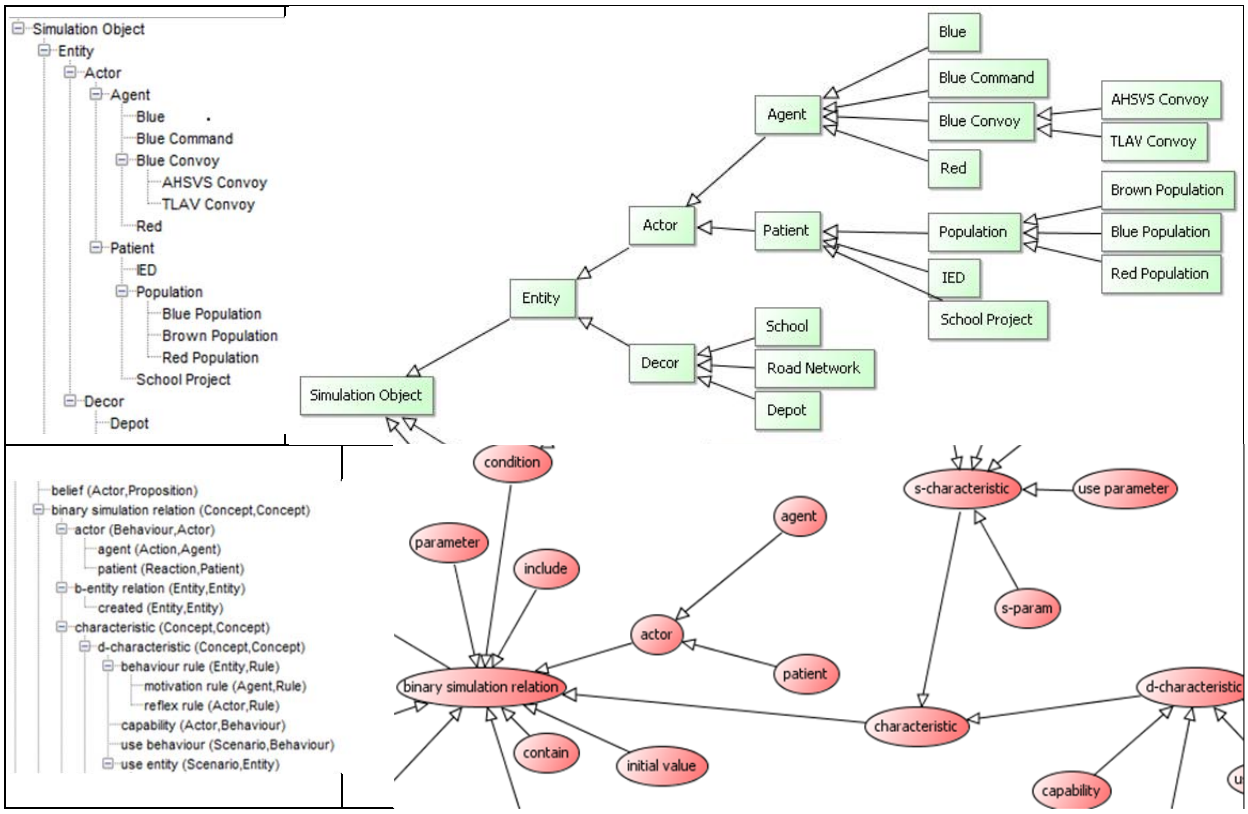

Figure 2: Specialization of the generic Scenarization Vocabulary

2.3 Agent-Based Modeling

The scenarization semantics includes all concept types relevant to building an agent-based model. The first step in building this model is to define the simulation entities, which are actors (agents and patients) and decors involved in the situation, along with their properties (static and dynamic). The entities, when put together, create the environment in which they evolve. Static properties provide simulation entities with fixed characteristics that cannot change during a simulation (e.g., the weight of an object, the maximum power of an engine). Dynamic properties, on the other hand, provide simulation actors with a mechanism to store their current state, which is a function of the various events occurring in the environment (e.g., the position of an individual, the current speed of a moving vehicle). In the current framework, properties can be of several types, ranging from simple (e.g., integer, double, string) to complex (similar to structures in modern programming languages), all represented by conceptual graphs.

The second step in building an agent-based model is to add capabilities to actors, each capability being associated with one behavior. Behaviors, which are activated at runtime by the simulation scheduler, manipulate properties and have the effect of adjusting dynamic properties, thus changing actor states.

At runtime, behaviors are scheduled according to a simulation cycle defined in the modeling framework. A cycle sequence is either fixed (i.e., behaviors always execute in the exact same order) or dynamic. Dynamic scheduling requires the definition of stimulations that activate actors' behaviors. There are two types of stimulations: *motivation* and *reflex*. A motivation will trigger an actor's behavior each time step that a condition is verified, whereas a reflex will trigger a behavior only when a change occurs in the environment (actor added or removed, or dynamic property value changed). Stimulations allow for a more dynamic behavior scheduling, managed via priorities, which in turn increases the flexibility of the modeling framework. Finally, the agent-based modeling framework includes an experimental feature: *predi-*

cates. Predicates allow a scenario designer to include some logic in the conceptual graphs. For the moment, they are used to define the conditions under which stimulations are enabled.

Figure 3 shows how an agent is specified with the conceptual graph formalism. CoGUI-IMAGE allows for the insertion of prototypic graphs, which automatically create agent templates. Inserting a prototypic graph also avoids potential syntax errors in conceptual graphs. Figure 3 shows the resulting definition of a "Blue Convoy" agent. It includes three capabilities ("Detect IED", "Advance on Route" and "Find Next Segment"); two stimulations (one associated with the "Detect IED" behavior, another with the execution of "Advance on Route" before "Find Next Segment") along with their predicates; several static properties; two static types ("TLAV Convoy" and "AHSVS Convoy": their parent-child relation with "Blue Convoy" is established by the simulation entity hierarchy (top of Figure 2); see 3.2.1 for the definition of static types); and several dynamic properties.

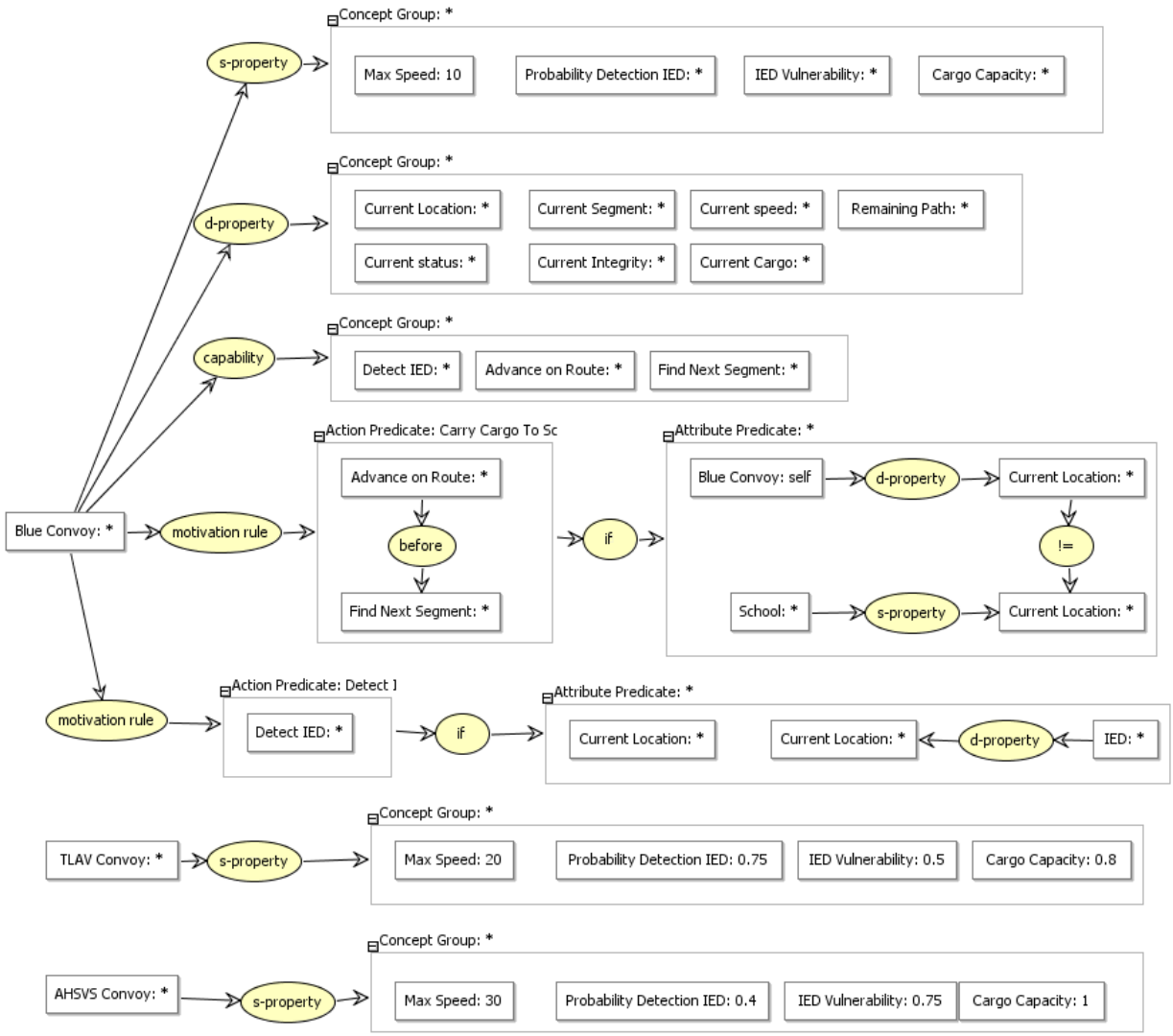

Figure 3: Specification of an Agent

3 FROM CONCEPTUAL MODELS TO SIMULATION EXECUTION

The process of creating conceptual graphs to express a situation in the agent-based formalism can require a large amount of work, depending on the complexity of the situation. It allows structuring how one views

a problem, and potentially helps in the comprehension process. The next step toward obtaining an executable simulation model consists of implementing the agents using the formalism of an agent-based simulator (e.g., MASON (Luke et al. 2005), Ascape (Parker 2001), and Repast Simphony (North et al. 2005)). In the current work, the large amount of information contained in the conceptual graphs is exploited to automatically generate executable models rather than forcing the designer to do this work himself. This eliminates potential transcription errors between the conceptual and simulation models, keeps both models synchronized, and allows for their straightforward iterative development.

The next section presents the process of transforming conceptual graphs into executable code (including a scenario that contains the initial state of the simulation). The generic agent-based simulation (GABS) framework is then presented, and the benefits of employing this method instead of traditional ones for models development are examined.

3.1 Transformation Process

As can be seen in Figure 4, three software components are involved in the generation and execution of simulation models. Firstly, a designer needs to represent models in CoGUI-IMAGE using the scenarization semantics introduced above. This step is crucial for the whole process to be successful because unstructured graphs are much more difficult to interpret than graphs that are expressed in a known format.

The second step is the automated process of transforming conceptual graphs into executable simulation models. The transformation takes as input all relevant conceptual graphs (i.e., scenario, actors, patients, behaviors), parses and interprets the graphs, asks the designer for his input should there be missing values, and writes the result to appropriate data structures. These are processed by two separate transformation pipelines (see the middle of Figure 4).

The first one generates executable models. It includes class descriptions, which define a generic class (in terms of object-oriented programming) as well as its attributes. This class description is then provided as input to a code generation template and results in Java code (class skeletons and management code generated by the Apache Velocity toolkit (Apache Foundation, 2011)). A template is available for each type of class file to be generated (static properties, dynamic properties, entity components, behaviors, motivations, reflexes and simulation cycles). A developer has to implement the dynamics of the models by filling in the skeleton classes of the cycles, behaviors and stimulations. The resulting source code is then compiled to Java byte code, ready to be loaded by the simulator (given an associated scenario).

In the second pipeline of Figure 4, a scenario based on the metamodel of the GABS framework is constructed. This structure contains all the information required to load the initial state of the generated scenario. For example, it contains the descriptions and contents of the static properties, the names and instance identifiers of the actors, their associated behavior, motivation and reflex classes, and a description of the cycles and cycle steps. Additional details about the GABS framework are provided later on. This structure is marshalled to an XML file using the JAXB (Fialli and Vajjhala 2003) marshaller implementation. Since the simulator's metamodel, expressed as an XML schema, is well-known and fixed, it is much easier to manipulate native classes than to compose an XML file programmatically. However, the downside of this approach is the difficulty of using an alternative simulator.

The rationale for separating the model implementations from the actual data is the enhanced flexibility that this lends to the entire transformation pipeline: it is good software engineering practice to separate the data model from the actual implementation. In addition, if one wants to share the models one implemented but one does not want to reveal the details of the implementation, this can be easily accomplished by sharing a compiled version of the models (withholding source code) while preserving the collaborators' ability to change the initial values of the static properties by editing the XML scenario.

Once scenario generation is complete, one proceeds to simulation. The generated simulator takes as input the XML scenario and dynamically loads the generated Java classes in order to form the initial simulation state. Then, the simulator's scheduler executes simulation cycles until an end condition is reached. It is also possible to interrupt the simulation and dump its current state to an XML file for later resumption. This operation requires that the dynamic properties serialize correctly; the required methods are gen-

erated during the code generation process. However, special care should be taken to implement stateless models only; otherwise, reloading a saved simulation state could result in incorrect execution when compared with a simulation that ran continuously. The whole annotated process can be seen in Figure 4, which highlights each step in blue rectangular boxes.

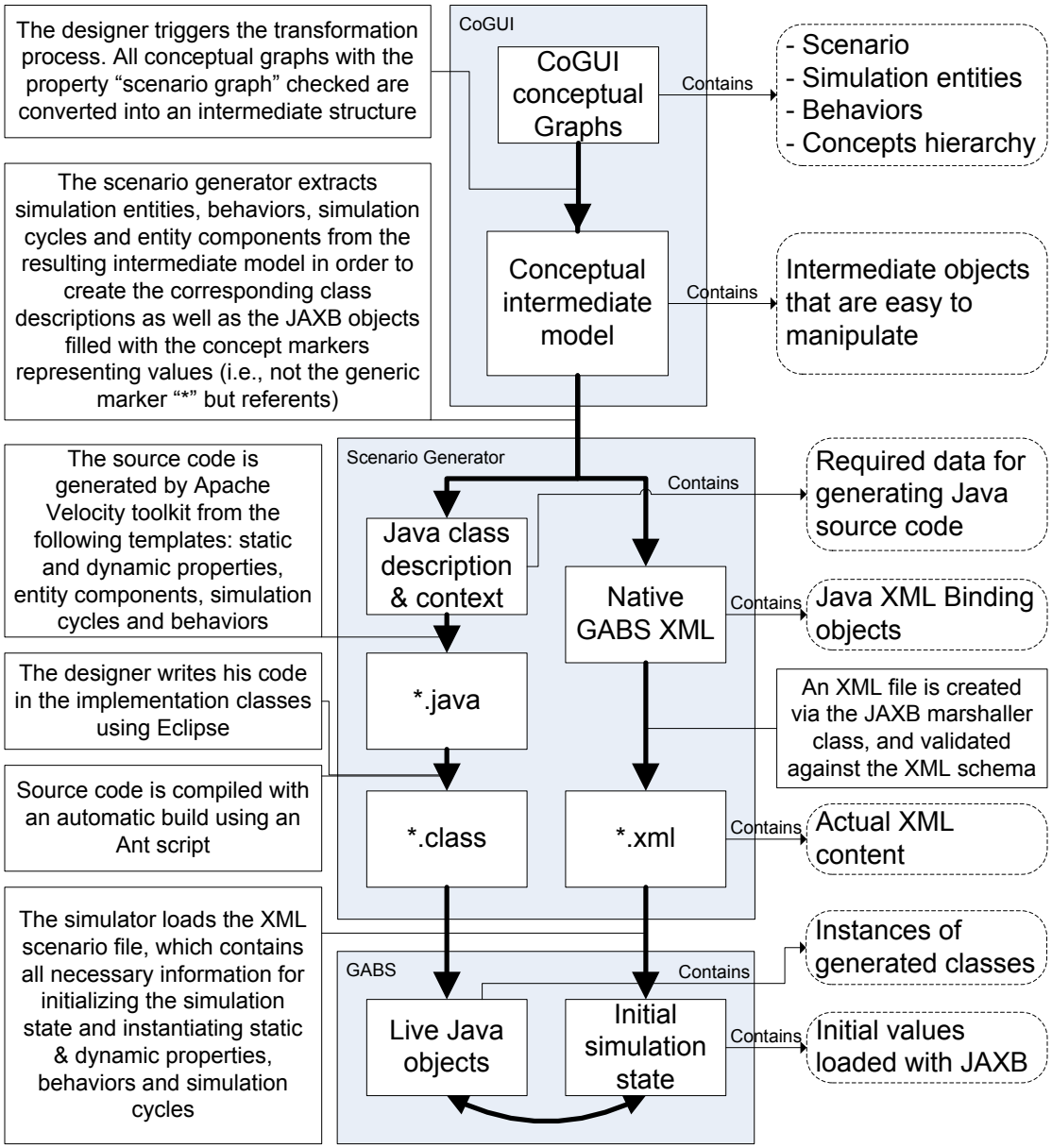

Figure 4: From concept to executable simulation

3.2 Generic Agent-Based Simulation (GABS) Framework

In order to support the simulation of agents that are described conceptually, a generic agent-based simulator framework was developed. GABS allows agents to interact with one another as a consequence of their behaviors, which are activated either proactively (motivations) or reactively (reflexes). In order for behaviors to be scheduled properly, a simulation cycle is defined that either activates behaviors in a predefined

order or based on stimulation priorities. The difference between the two time-stepped scheduling modes is not cosmetic. For instance, several behaviors can be associated with a single stimulation, and the same behavior can be associated with more than one stimulation. The stimulation mechanism is thus more flexible than fixed behavior scheduling.

3.2.1 GABS Metamodel

GABS relies on a metamodel that defines all the entity types that a simulation can contain, along with the technical details required for the good functioning of the simulation. The scenario's metamodel, which is expressed as an XML schema to be compiled by the JAXB library, has the following structure:

- Setup: Contains the generic scenario parameters that are accessible to every user-defined method;
- Agents: A collection of object classes, each with the following properties:
  - Type: a generic type identifier. Does not need to be unique (see Type ID), thus allowing several agents of the same type to be defined, each having its own static properties;
  - Type ID: a unique identifier for this type of agent (and its Static Types);
  - Parent Type ID: a reference to the type ID for the parent type of this agent. When an agent has a parent, it inherits its properties (static and dynamic) as well as its capabilities;
  - Instances: provides the names of the agent instances that are present in the environment at the beginning of the simulation;
  - Static Properties: contains typed data structures associated with the agent. Static properties are either of simple (native types, e.g., integer, floating-point and double-precision numbers, character strings) or complex types (compositions of simple and complex types). They are unchanging and common to all instances of the agent;
  - Static Types: are sibling types of the current one, that differ only in that they use a different set of static properties;
  - Behaviors: A collection of distinct behaviors, each with the following properties:
    - Name: the name of the behavior. These behaviors are shared by every agent instance of the current and descendant types. Thus, the implementation must not be agent instance-specific: behaviors should consult the agent instances' dynamic properties instead;
    - Behavior Class: the name of the Java class that implements this behavior;
    - Parameters List: a static list of parameter values passed to the behavior's implementation (this allows a given Behavior Class to be used to implement several different behaviors);
    - Sub-behaviors: an ordered list of behaviors that are executed before the effects of the current behavior are applied;
  - Reflexes: A collection of reflexes, each with the following properties:
    - Name: the name of the reflex. Note that as for behaviors, reflexes are shared among actor instances of the same type and inherited by descendant types;
    - Stimulation Class: the name of the Java class that triggers this reflex;
    - Behaviors: an ordered list of behavior names that are executed when this reflex is activated, taken from the agent's Behaviors;
  - Motivations:
    - Name: the name of the motivation. As for behaviors and reflexes, motivations are shared among actor instances of the same type and inherited by descendant types;
    - Stimulation Class: the name of the Java class that triggers this motivation;
    - Behaviors: an ordered list of behavior names that are executed while this motivation is activated, taken from the agent's Behaviors;
- Patients:
  - This collection of objects has the same properties as the Agents with the single exception of the Motivations which are absent;
- Decors:

  - This collection of objects has the same properties as the Agents except that Static Types, Behaviors, Reflexes and Motivations are absent;
- Execution:
  - Cycle (recursive definition due to the sub-cycles):
    - Name: name of the cycle;
    - Class: the name of the Java class that implements this cycle;
    - Priority: the priority of this cycle's Class with respect to sibling schedulable entities;
    - Cycles (zero or many): sub-cycles that execute inside this cycle, along with their priorities. They are typically used in simulations that involve several levels of detail for the models;
    - Scheduled Behaviors (zero or many): references to behaviors along with their priorities as well as the actor IDs (references to instances) or the actor types they affect;
    - Scheduled Stimulations (zero or many): references to stimulations along with their priorities as well as the actor IDs (references to instances) or the actor types they affect;
    - Parameters List: a static list of parameter values that are available to the user-defined methods of this cycle;
- Simulation (optional): when present, it signals a simulation in progress (a saved version of the simulation);
  - Iteration: the current iteration number;
  - Random Number Generator: a saved version of the random number generator that is used to add stochastic effects to a simulation;
  - Live Actors: the list of actors that are instantiated in the saved simulation state:
    - TypeID: the actor type ID;
    - Name: the actor instance name;
    - Static Type: the static type (to disambiguate between types sharing a single Type ID);
    - Live Properties: the saved version of dynamic properties;
    - Live Active Behaviors: a list of behaviors that are active at the time the simulation state is saved. Needed because behaviors can span several simulation time steps;
  - Live Execution: a saved version of the simulation cycle, which includes the iteration, next element in the execution sequence, activated reflexes;
    - Live Parameters: the cycle's current parameter values.

3.2.2 Important Features of GABS

Several important features not yet described can be highlighted simply by looking at the metamodel introduced above. Firstly, the scenario allows for the definition of *global parameters* that are accessible from every implementation method. Those are to be placed in the actual "Scenario" conceptual graph, and are included in the "Setup" part of a scenario instance.

As can be seen in the GABS metamodel, agents, patients and decors contain a "ParentType ID" field. It allows for the definition of a *hierarchy of entity types* (top of Figure 2). A child type will share parent properties (static and dynamic) as well as capabilities (if they are actors). Dynamic property values are owned separately by each instance, whereas static property values, being unchanging, are shared throughout the pertinent portion of the type hierarchy. A child type needs to be defined in its own conceptual graph. In the transformation process, the establishment of a hierarchy of entity types is performed at the conceptual level, in the concept types hierarchy. When the conceptual graph interpreter encounters a concept hierarchy, it generates appropriate source code, and uses the inheritance and polymorphism features of the Java object-oriented programming language in order to manage child and parent types as well as static and dynamic properties appropriately.

Agents, patients and decors also contain an "Instances" field that defines entity *instances that will initially be part of the simulation*. Those are optional, but when present remove the need for creating entity instances programmatically either in the cycles or behaviors. When the context allows for knowing in ad-

vance which actors are part of a simulation, another advantage of defining them in advance is their facilitated accessibility. Indeed, a number of Java classes and interfaces are generated during the transformation process that allow for straightforward access to initial instances (of the form "EntityType_EntityName") by all implementable methods of simulation cycles and behaviors.

The "Static Types" field allows for the definition of several *entity types* that share the same stimulations, behaviors, static and dynamic property types without needing to specify an entire conceptual graph. "Static typing" differs from defining a child type because new properties or behaviors may not be declared: one may *only* declare a different set of static property values. It is a usability shortcut while building conceptual graphs. Figure 3 show how static types are typically defined in conceptual graphs (see "TLAV Convoy" and "AHSVS Convoy").

For several simulation objects (behaviors, stimulations and cycles), one parameter declares the *Java class name that implements the actual object*. Those class names are generated during the transformation process and should not be changed by the developer unless he wants to add functionalities to a class whose source code is not available. Note that classes associated with static and dynamic properties are also loaded by GABS at runtime, but the actual class names are composed by the simulator. GABS also allows "Observers" (omitted from the metamodel description for clarity) to be part of the scenario, neutral objects which do not act upon the simulation but are notified of the simulation state at every time step. These hooks into the simulation could be used to plug visualization software or hook external learning or monitoring algorithms to the simulation.

Finally, GABS allows for *checkpointing and reloading a simulation state*. The information associated with a simulation state is saved in a "Simulation" object which includes notably the value of all the actors' dynamic properties, the state of the current active behaviors, as well as the state of every running simulation cycle along with its dynamic parameters. GABS has been integrated into Multichronia, a software which allows for interactive simulation (Rioux, Bernier, and Laurendeau 2008). Using Multichronia, a user can explore a parameter space and better understand how his/her models behave when modifying parameters.

3.3 Model Development

Most of the Java code generated during the transformation process is complete and does not need any additional work by the designer, although some may. The designer will find in the generated packages a set of implementation classes, including method stubs, used mainly to detail relevant simulation dynamics. These are behavior, stimulation and simulation cycle classes. Although there are plans to improve graph notation and minimize code writing by the developer, this stub approach was chosen deliberately for two main reasons. First, it allows for a maximum of flexibility, enabling any code insertion of interest by the developer. Secondly, the graph notation exists to ease the user's work and, in many cases, a behavior may be more easily expressed using a programming language rather than a graph notation. A detailed list of the methods that need to be filled by the developer can be found in Lizotte and Rioux (2010).

While conceptualizing the transformation process, special care was taken to ease as much as possible the developer's burden of model implementation. For that purpose, several helper classes and methods are generated at the same time as the actual executable code. For example, it is possible to access every instance of an actor type via a static method ("get"), which returns a collection of instances. In addition, all the generated code is strongly typed by making use of the "generics" feature of Java. The latter is possible because types are mandatory in the conceptual graphs as well. For example, if the designer defines a property as being a floating-point number in CoGUI-IMAGE, the same property will be of type "float" when he accesses it in behaviors via the getter and setter methods.

Finally, the current framework encourages one to develop models iteratively. Properties can be added and removed without harm: when a property is added, it is accessible through the getter/setter methods of an agent; when it is removed, the getter/setter methods are no longer available and any remaining usage instances will result in compilation errors. With this approach, errors are detected at compile time, not at

run time. It prevents a lot of errors while keeping the flexibility of a generic agent-based modeling environment.

4 RELATED WORK

Agent-based modeling has been a popular topic in the simulation community for a long time (Macal and North 2006). Several frameworks have been developed that address part of the problem but none deal with the comprehension aspect of the models as it is presented in this paper (Luke et al. 2005, Parker 2001, Parker 2011, North et al. 2005). The agent modeling platform (AMP) (Parker 2011) is closely related to our work because it allows one to express agents using a metamodel (similar to our scenarization semantics) and generates executable models that are Java skeletons and interfaces. Repast Simphony (North et al. 2005) also allows modeling the dynamics of the agent behaviors using a visual interactive interface. However, we go even further in modeling the agents from a conceptual point of view, letting a designer first express his knowledge without the simulation constraints and later align his knowledge to the simulation ontology in an integrated environment.

5 PRELIMINARY RESULTS, FUTURE WORK AND CONCLUSION

Two completely different models using the current framework were implemented successfully. The first one implements a series of military convoys circulating on a road network in order to carry goods from a depot to a school construction site. These convoys are attacked by improvised explosive devices (IED) placed by insurgents on road segments. The quantity and strength of IEDs are a function of the population allegiance that is influenced by the passing convoys as well as the explosions. This scenario has been integrated with Multichronia (Rioux et al. 2008) and the IMAGE visualization tools (Mokhtari et al. 2011).

The second scenario uses the framework to implement a serious game simulating a system dynamics model. It is part of the complex human decision making experimentation (CODEM) platform (Lafond and DuCharme 2011). The model consists of variables and action points that a user can allocate in order to influence the situation. This application, which makes extensive use of the actors inheritance feature of the framework, confirmed that the approach is generic and really independent of the application domain. In other words, it can be specialized to a huge family of problems.

Even though the framework was successfully used for two very different problems, it is far from perfect. The CoGUI-IMAGE software has several limitations in terms of usability and stability. In addition, it does not feature a metamodel that would facilitate the extraction of conceptual graphs and their transformation using frameworks more advanced than Apache Velocity. Therefore, building an improved conceptual graphs modeling software is part of the future work. In addition, it could be integrated in the Eclipse platform, which is already used to develop the Java code and launch the scenarios.

Finally, a major limitation of our prototype is its lack of integration with commonly used agent-based simulators. The implementation of our approach using either MASON, Ascape, Repast Simphony, or others would prove that our framework is general and would benefit the agent-based simulation community.

REFERENCES

Apache Foundation. 2011. *The Apache Velocity Toolkit*. Accessed March 31. http://velocity.apache.org/.

Chein, M., and M. L. Mugnier. 2008. *Graph-Based Knowledge Representation: Computational Foundations of Conceptual Graphs*. London: Springer-Verlag.

Fialli, J., and S. Vajjhala. 2003. *The Java Architecture for XML Binding (JAXB)*. Sun Microsystems.

Genest, D. 2010. *Cogitant Reference Manual version 5.2.3*, Accessed March 31. http://cogitant.sourceforge.net/files/cogitant.pdf.

Lafond, D., and M. B. DuCharme. 2011. "Complex Decision Making Experimental Platform (CODEM): A Counter-insurgency Scenario." In *Proceedings of the IEEE Symposium on Computational Intelligence for Security and Defence Applications,* 72-79. Paris.

LIRMM (Laboratoire d'informatique, de robotique et de microélectronique de Montpellier). 2011. *Co-GUI: A Conceptual Graph Editor*. Accessed July 13. http://www.lirmm.fr/cogui/.

Lizotte, M., and B. Moulin. 1989. "SAIRVO: A Planning System which Implement the Actem Concept." *Knowledge-Based Systems Journal* 2(4):210-218.

Lizotte, M., D. Poussart, F. Bernier, M. Mokhtari, E. Boivin, and M. B. DuCharme. 2008. "IMAGE: Simulation for Understanding Complex Situations and Increasing Future Force Agility Models." In *Proceedings of the Army Science Conference 2008*. Orlando, FL.

Lizotte, M., and F. Rioux. 2010. "IMAGE-Scenarization: A Computer-Aided Approach for Agent-Based Analysis and Design." In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 837-848. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Luke, S., C. Cioffi-Revilla, L. Panait, K. Sullivan, and G. Balan. 2005. "MASON: A Multi-Agent Simulation Environment." *Simulation: Transaction of the Society for Modeling and Simulation International* 82(7):517–527.

Macal, C., and M. North. 2006. "Tutorial on Agent-Based Modeling and Simulation Part 2: How to Model with Agents." In *Proceedings of the 2006 Winter Simulation Conference*, edited by L. R. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, 73–83. Piscataway, NJ: Institute of Electrical and Electronics Engineers, Inc.

Mokhtari, M., É. Boivin, D. Laurendeau, S. Comtois, D. Ouellet, J.-C. Levesque, and E. Ouellet. 2011. "IMAGE – Complex Situation Understanding: An Immersive Concept Development." In *Proceedings of IEEE Virtual Reality Conference*, 229-230. Singapore.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence*. Palo Alto, CA: Tioga Pub. Co.

North, M. J., T. R. Howe, N. T. Collier, and J. R. Vos. 2005. "The Repast Simphony Development Environment." In *Proceedings of the Agent 2005 Conference on Generative Social Processes Models and Mechanisms,* 151–158. Chicago, IL.

Parker, M. 2001. "What is Ascape and Why Should You Care?" *Journal of Artificial Societies and Social Simulation* 4:1.

Parker, M. 2011. *Agent Modeling Platform*. Accessed March 31. http://www.eclipse.org/amp/.

Peirce, C. S. 1933. *Collected Papers of Charles Sanders Peirce*, Vol. 4, *The Simplest Mathematics*, Book II: *Existential Graphs*. Cambridge, MA: Harvard University Press.

Rioux, F., F. Bernier, and D. Laurendeau. 2008. "Multichronia – A Generic Parameter, Simulation, Data, and Visual Space Exploration Framework." In *Proceedings of the Interservice/Industry Training, Simulation & Education Conference*. Orlando, FL.

Sowa, J. 1984. *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison Wesley.

AUTHOR BIOGRAPHIES

FRANÇOIS RIOUX received a B.Eng. Degree in Electrical Engineering from Laval University in 2003. He received an M.Eng. Degree from McGill University in 2005. He received his Ph.D. in Electrical Engineering from Laval University in 2009. He completed the work presented in this paper as a consultant for the LTI Software and Engineering firm. He now works as a software architect for Thales Canada - Systems Division. He performs research in collaboration with Defence R&D Canada – Valcartier on the topics of interactive simulation and visualization applied to the better understanding of complex systems. His e-mail address is francoisrioux@gmail.com.

MICHEL LIZOTTE has been working at Defence Research and Development (DRDC) – Valcartier since 1999. He is currently the Group Leader of the Simulation and Comprehension of Complex Situations group. Previously, he was an information technology (IT) consultant for almost 11 years including eight at DMR. During this period, the vast majority of his assignments were carried out for research es-

tablishments such as the National Research Council (NRC) of Canada. He obtained his Bachelor's Degree in Computer Science (1984) and his Master's Degree in Artificial Intelligence (1988) from Université Laval (Québec). His current interests and recent work encompass approaches to understanding complex situations and software-intensive system architecture, software and military capability engineering, and software and system architectures. His e-mail address is Michel.Lizotte@drdc-rddc.gc.ca.