A Framework for Modeling and Simulation of the Artificial

Scott A. Douglass and Saurabh Mittal

Abstract *Artificial systems* that generate contingency-based teleological behaviors in real-time, are difficult to model. This chapter describes a modeling and simulation (M&S) framework designed specifically to reduce this difficulty. The described Knowledge-based Contingency-driven Generative Systems (KCGS) framework combines aspects of SES theory, DEVS-based general systems theory, net-centric heterogeneous simulation, knowledge engineering, cognitive modeling, and domain-specific language development using meta-modeling. The chapter outlines the theoretical and technical foundations of the KCGS framework as realized in the Cognitive Systems Specification Framework (CS2F), a subset of KCGS. Two executable models are described to illustrate how models of autonomous, goal-pursuing cognitive systems can be modeled and simulated in the framework. The technical content and agent descriptions in the chapter illustrate how the *M&S of the artificial* depends critically on *ontology*, *epistemology*, and *teleology* in the KCGS framework.

1 Introduction

This chapter describes the Cognitive Systems Specification Framework (CS2F), as a subset of Knowledge-based Contingency-driven Generative Systems (KCGS) framework; a modeling and simulation (M&S) framework designed to support the M&S of models, agents, and cognitive systems capable of autonomously designing their own behavior in real-time. The CS2F framework is based on advances made

Scott A. Douglass, PhD
Air Force Research Laboratory, 711 HPW/RHAC, 2620 Q Street, Bldg 852, Rm 3-331, WPAFB, OH 45433-7955 e-mail: scott.douglass@wpafb.af.mil

Saurabh Mittal, PhD
L-3 Communications, 2620 Q Street, Bldg 852, Rm 3-327, WPAFB, OH 45433-7955 e-mail: saurabh.mittal@l-3com.com

in System Entity Structure (SES) theory [43], the Discrete Event Systems (DEVS) Unified Process [18], and large scale cognitive modeling (LSCM) research initiative [5, 20, 21].

The chapter begins with a discussion of a modeling problem the framework is intended to solve. The problem boils down to the present difficulty of modeling and simulating autonomous cognitive systems. The CS2F framework is intended to decrease this difficulty. Section 2 describes *artificial systems* and the artificial phenomena they produce. This section argues that autonomous models and agents are difficult to specify because: (1) they produce artificial phenomena; phenomena that reflect contingencies, choice, and teleology; (2) current modeling frameworks lack comprehensive support for the formal specification of relationships between contingencies, choices, and goals. Section 3 presents the CS2F framework and discusses the modeling formalisms and net-centric simulation technologies that constitute it. This section describes the framework as a componentized environment in which artificial phenomena can be readily modeled and simulated. After describing the framework, the chapter illustrates how models of artificial phenomena are actually specified and executed. Section 4 describes two agents: one that learns to adjust its behavior to match the probability structure of the environment; and another that uses abduction, a type of inference that refines knowledge, to make sense of its situation. While these agents are simple to facilitate exposition, they clearly demonstrate how the CS2F framework is used to model and simulate artificial phenomena. In Section 5, the broader theoretical background and ambition of the KCGS framework are presented. In Section 6, the chapter finally argues that the framework's effectiveness can be traced to the integration of aspects of *ontology*, *epistemology*, and *teleology* into modeling and simulation.

1.1 The Problem

Cognitive scientists employing computational process modeling in their research consider cognitive activity to be a product of an open system that interacts with the environment [41]. This perspective has motivated many cognitive scientists, especially those in the information processing psychology and cognitivist research traditions, to study *cognitive architecture*, the structural and behavioral system properties underlying cognitive activity that remain constant across time and situation. The Adaptive Character of Thought-Rational (ACT-R) is a theory of human cognition in the form of a cognitive architecture [2]. While cognitive modeling frameworks such as ACT-R allow cognitive scientists to explain/predict activities and processes occurring within an invariant architecture, they say little about how the influences of situational contingencies in which cognition occurs should be formally captured and related to behavior.

Let a *program* be sequence of instructions that perform a task when executed. Let an *agent* be something that perceives and acts. Agents can act on the basis of: (1) contingencies; (2) built-in knowledge; or (3) a combination of contingen-

cies and built-in knowledge. Let an *autonomous agent* be one that bases its actions on its contingencies. Let *acting rationally* be given beliefs, acting so as to achieve goals. Cognitive scientists are struggling to develop cognitive models that behave more like autonomous rationally acting agents than programs. To be autonomous, an agent must base its actions on the constraints and affordances of its situation. A key consequence of this dependence on situational factors is that the contingencies an autonomous agent acts in (factors outside the invariant architecture) play as important a role in determining its actions as its invariant architecture. Autonomous rational agents are difficult to develop because the broader system in which contingencies and cognitive activity mesh must be represented and processed by the agent.

1.2 The Solution

AFRL efforts to resolve the above problem have produced a Cognitive Systems Specification Framework (CS2F), as a subset of the proposed KCGS framework. CS2F combines modeling formalisms (or Domain Specific Languages) in which models of systems that produce artificial phenomena can be specified. The KCGS framework provides a metamodel-based computing infrastructure wherein the CS2F modeling formalisms can be formally anchored in DEVS component-based systems specifications and ultimately simulated. The following aspects of the KCGS framework execute in concert to computationally realize the modeling and simulation of artificial systems as realized in CS2F:

1. Domain specific languages (DSLs) allowing modelers to formally specify the structure of knowledge related to; the environment, agent behaviors, states, goals, and domain theories. These DSLs allow domain experts to provide collaborative input in a larger systems context wherein heterogeneous components and multiple implementation platforms are the norm.
2. DSLs that use hierarchy to manage complexity in systems consisting of a large number of entities. These DSLs capitalize on formal properties of DEVS related to closure under coupling and the formal systems specification of hierarchy.
3. DSLs that use domain abstractions to limit specification and computational complexity during the specification and simulation of artificial systems.
4. Model-to-model transformation technologies that formally transform model component specifications in DSLs into executable DEVS systems-models. These executable models combine modeled aspects of both agents and their environments.
5. Knowledge processing mechanisms that refine knowledge while the system is in operation. These processes occur during simulation and allow agents to generate effective action and learn.
6. Capabilities based on variable structure modeling that support structural change in the system in operation. These capabilities change an agent's behavioral repertoire so that it reflects the dynamism and contingencies of the environment.

7. Capabilities based on event-based modeling that inject new knowledge into an autonomous agent at runtime. These capabilities support the learning and adoption of new knowledge.

The CS2F framework is designed to allow modelers to combine state, goal, and domain knowledge in cognitive domain ontologies (CDO) and simulate artificial phenomena in DEVS. These abilities allow cognitive scientists to model and simulate cognition as an artificial phenomenon contingent on situational factors, guided by a library of cognitive capacities (or behavior repertoire) and runtime constraints that operate between the agent and its environment.

2 Artificial Systems

In a review of embodied cognition, Wilson [41] proposes that science should study systems that are essentially permanent in structure; systems whose behaviors are invariant across situations. Underlying Wilson's proposition is a notion that if science is to understand and predict systems and processes in nature it must focus on, and model, fundamental principles of organization and function, not the behaviors of systems in specific situations. Put another way, when the specification of a scientific model appeals to situation-specific factors, scientists cannot predict the behavior of the model when the situation changes. Wilson illustrates the importance of focusing on the invariant aspects of studied systems by pointing out how scientific understanding of hydrogen is based on fundamental understanding of atoms, not understanding of how hydrogen behaves in a large number of contexts. Wilson suggests that scientists working to understand how cognition is *situated* or *embodied* are straying from a preferred focus on system behaviors that are invariant across situations. Put another way, rather than studying cognitive activity in specific situations, cognitive scientists should study cognitive architecture, the invariant structural and behavioral system properties underlying cognitive activity that remain constant across time and situation.

2.1 Artificial Phenomena

The "Achilles' heel" of Wilson's proposition is the obvious fact that human behavior is quite different from the behavior of hydrogen. Simon [35] draws out this difference by contrasting natural phenomena with artificial phenomena. *Natural phenomena* (for example, the behavior of hydrogen) are based on necessity; the behaviors of systems producing natural phenomena are subservient to natural law. *Artificial phenomena* (for example, the goal-pursuing actions of a human) are based on contingency; the behaviors of systems producing artificial phenomena are improvisational and reflect choices and requirements. Cognitive scientists endeavoring to model autonomous models and agents should develop theories and methods that enable them

to understand cognition as an artificial phenomenon profoundly influenced by situational factors and contingencies. This chapter illustrates how knowledge about situational factors and contingencies can be represented in ontologies and processed by teleological (goal-pursuing) agents in order to refine their knowledge and gain real-time behavioral autonomy.

2.2 State and Process Descriptions

Simon [35] argued that two representations of knowledge underlie artificial human behavior: *state description* knowledge and *process description* knowledge. Humans generate/design effective behaviors by posing and solving problems that link their goals to the actions they can take. They pose the problems by clarifying state descriptions of their goals and then solve the problems by discovering sequences of actions or processes that produce their goal states. We propose that for an agent to act autonomously, it must be able to convert state descriptions (representations of goals) into process descriptions (representations of actions). Specifically, an agent must be able to determine: (1) ***what*** actions are likely to achieve goals; and (2) ***how*** to perform these actions.

While it may be straight forward to specify how an agent is to perform actions using sequences of instructions (a program), it is extremely difficult to specify how an agent is to autonomously determine what it should do in its circumstances. This difficulty is partly due to the way answering the ***what*** question takes place *in situ*; in the confluence of situational constraints, current goals, perceived possible actions, cognitive limitations, preferences, etc. It is virtually impossible to specify all the ways contingencies shape actions in a model consisting of built-in rules. To develop autonomous agents that pursue goals in unpredictable environments, cognitive modelers need modeling formalisms and frameworks with which they can specify and execute models and agents that "soft-assemble" their actions. These formalisms and frameworks must allow modelers to separate the ***what*** and ***how*** concerns so that answers to the ***what*** question can be used to assemble sequences of instructions or rules that answer the ***how*** question *in situ*. This chapter describes formalisms and an execution framework meeting these requirements.

2.3 Modeling Artificial Systems

Modeled as intelligent artificial system, autonomous agents pursue goals while interacting with the environment and dealing with their contingencies. Such an autonomous agent must be able to situate itself in the environment, perceive the constraints and affordances of the moment, and relate contingencies to goals in order to take effective action. In order to design such an autonomous artificial system, the Modeling and Simulation discipline must be able to formally specify the structure

of the domain of the agent. Unless we formally specify the entire agent/contingency environment, the behavior specification of the agent will be fragile in the face of events occurring in the environment not anticipated by pre-defined rules. This implies that we should model the *whole* environment and a *rich* behavior representation of such an agent situated in it. This certainly is computationally prohibitive and better methods of managing such information are being developed by the chapter authors.

3 CS2F Framework for Modeling and Simulating Artificial Systems

This section describes technologies and a framework for specifying, modeling and simulation of artificial systems. It addresses concepts like model interoperability, model transparency, domain specific languages, formal ontology representation, knowledge engineering, search mechanisms and their integration aspects. We begin this section by describing *meta-modeling* as a necessary aspect of model interoperability. Meta-models are abstract models of the domain of interest and result in a set of rules that define a "domain-model". Performing model transformations at the meta-modeling layers paves way for model integration and collaborative development. In the next subsection, we will discuss how meta-models are transformed to the DEVSML stack to make them executable. It should be noted that the models are not "executable" by design at the meta-modeling layer. They have to be transformed to a framework (DEVS in this case) that takes models and makes them executable (as a simulation). This separation of concerns is a central theme in DEVS based modeling and simulation that separates the modeling and simulation layers using a simulation relation.

3.1 Foundations of the CS2F Framework

3.1.1 Meta-Modeling

A domain specific language (DSL) is a dedicated language for a specific problem domain. For example, HTML is a DSL for web pages, Verilog and VHDL are DSLs for hardware description. A DSL can be can have a textual, graphical, or a hybrid concrete syntax and is essentially a meta-model of allowable specifications. A DSL exploits abstractions so that the respective domain experts can specify their problem without paying much attention to the general purpose computational programming languages such as C, C++, Java, etc. which have their own learning curve. In our efforts, we employ the Generic Modeling Environment (GME) as a DSL development framework. GME is a highly configurable meta-modeling environment developed by the Institute for Software Integrated Systems (ISIS) at Vanderbilt

University [13, 14, 28, 38]. The GME is essentially a tool for creating and refining domain-specific modeling and program synthesis environments. GME meta-models are specified using a graphical/textual notation resembling UML class diagrams.

GME has been used by the authors to develop the Cognitive Systems Specification Framework (CS2F), a composition of domain-specific languages tailored to the requirements of specifying models and agents that base goal-pursuing behaviors on contingencies. The DSLs currently composed in CS2F are:

CS2F/DM A specification language based on the OWL [17] ontology standard used to specify an agent's propositional or declarative knowledge.

CS2F/CDO A specification language based on SES theory used to specify models of domain knowledge combining aspects of agents and the situational factors or contingencies constraining their behavior.

CS2F/BM A specification language based on behavior models (predicated nondeterministic finite state machines) used to specify an agent's behavioral repertoire.

These three CS2F specification languages have been composed into a single meta-model defining an integrated authoring environment in which a modeler can specify the declarative, domain, and procedural knowledge of an agent.

3.1.2 Representations of State, Goal and Domain Knowledge

System entity structure (SES) theory is a formal ontology specification framework that captures system aspects and their properties [43]. In the past, SESs were used in design and simulation environments to formally capture configurations of systems that achieve a common design objective [11, 31–33, 42, 44].

In the early 1990s, researchers working at the overlap of artificial intelligence and modeling and simulation began to design and implement environments that automated the process of design space exploration [31] to solve engineering problems [32]. SESs were used to represent system configuration alternatives in these environments. The SES was primarily used to specify the relations between these entities [33]. In addition to capturing aspects, entities, taxonomic relationships, variable values, and structural/configuration alternatives, these SES included information about how entities in the SES could be realized in the DEVS formalism and composed into an executable model. To systematically explore design spaces in these environments: (1) rule-based search processes were used to derive all valid pruned entity structure (PES) captured by the SES; (2) information based on entities and aspects in each PES was used to compose an executable model using DEVS components stored in a model repository; (3) each composed model was simulated; and (4) simulation results were analyzed in order to identify the most desirable design alternative. The Solutions set is determined by the pruning process on the SES and the optimal solution was determined by simulation of each of the designs in Solutions set.

Rather than being used to capture system alternatives to be explored through DEVS-based modeling and simulation, SES are currently being used to formally capture structural and relational information about domains. SES are being used to specify entire ontologies rather than just system configurations that solve engineering problems [15, 16, 43]. This current use exploits similarities between SES and general ontologies. Current research and modeling and simulation activities utilizing SES demonstrate that extraordinarily diverse domains can be formally captured and related to each other through formal structures such as domain ontologies, pragmatic frames, and overlapping pruned entity structures (PES).

The CS2F framework described in this chapter uses cognitive domain ontologies (CDOs), a theoretical extension of SES to represent *spaces of behavior* as if they were system configurations. Situational/agent properties, aspects and constraints can be formally captured in CDOs. CDOs are processed by an agent to determine ***what*** it should do. The framework constitutes a modeling architecture that explicitly supports the representation and processing of CDOs. This capability allows modelers to separate the ***what*** and ***how*** concerns and specify agents that generate process descriptions by using answers to the ***what*** question to identify and "soft-assemble" knowledge into contextually appropriate process descriptions.

3.1.3 DEVSML 2.0 and the DEVS Unified Process

Discrete Event System Specification (DEVS) [46] is a formalism which provides a means of specifying the components of a system in a discrete event simulation. The DEVS formalism consists of the model, the simulator and the experimental frame as shown in Fig. 1. The Model component represents an abstraction of the source system using the modeling relation. The simulator component executes the model in a computational environment and interfaces with the model using the simulation relation or the DEVS simulation protocol in the present case. The Experimental Frame facilitates the study of the source system by integrating design and analysis requirements into specific frames that support analyses of various situations the source system is subjected to.

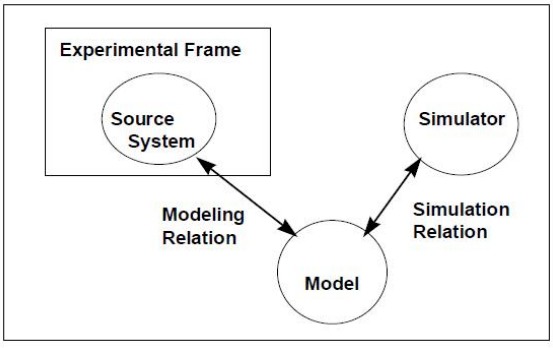

Fig. 1 DEVS Framework elements

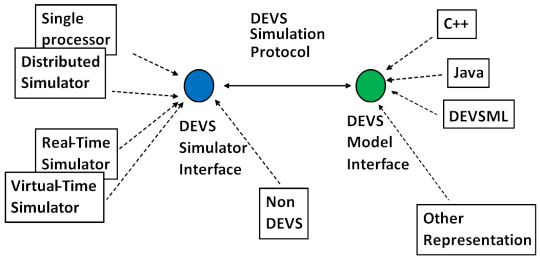

Fig. 2 Standardizing the Model and Simulator interfaces

While historically models have been closely linked to the platform (such as Java, C, C++) in which the simulator was written, recent developments in platform independent modeling and transparent simulators [25–27, 30] have allowed the development of both the models and simulators in disparate platform. To facilitate interoperability, integration and composability, a layered DEVS Modeling stack was proposed that executes on Service oriented Architecture (SOA) [24, 26]. Current efforts are focusing on a standardization process [39] wherein the simulation relation can be standardized for further interoperability [27, 45].

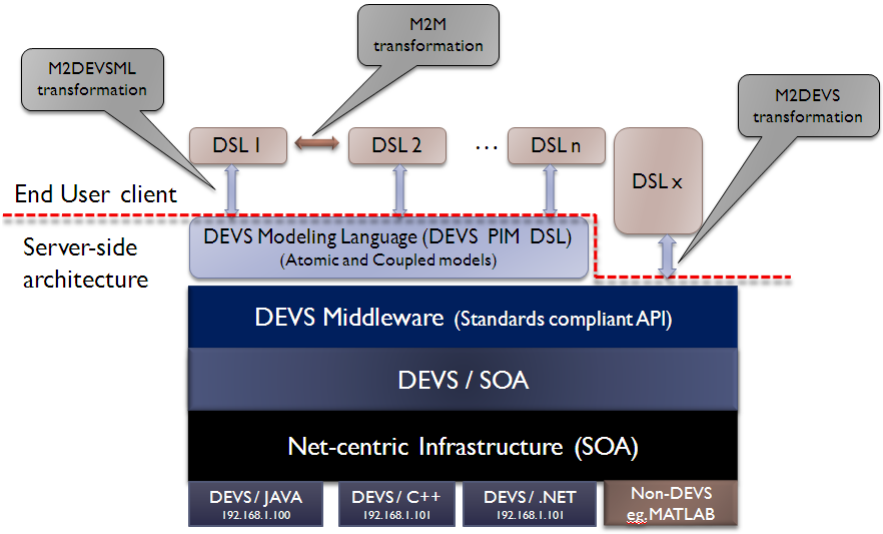

Fig. 3 DEVSML 2.0 stack enabling model and simulator interoperability

The latest version of this stack, shown in Fig. 3, was proposed as a part of Air Force Research Laboratory's Large Scale Cognitive Modeling (LSCM) initiative [5, 20, 21]. While the earlier version of the DEVSML stack was designed to provide XML interoperability and the netcentric transparent simulation to the DEVS models, the current version was designed to enhance scope and model interoperability [22]. Models specified in the new DEVSML 2.0 stack are specified in domain

specific languages (DSLs) and then through transformations are taken to the DEVS framework. The idea of accommodating suites of DSLs at the top layer of the stack is a major addition in the DEVSML 2.0 stack.

The DEVSML stack has been an integral component of the larger DEVS Unified Process (DUNIP) [18, 23]. DUNIP is a universal process and is applicable to multiple domains. However, the understated objective of DUNIP is to incorporate discrete event formalism as the binding factor at all phases of this development process. The important concepts, the processes within DUNIP and how they relate to CS2F are listed below:

1. **Requirements and Behavior specifications using Domain Specific Languages (DSLs)**: We mentioned CS2F/BM, CS2F/DM, and CS2F/CDO as DSLs that are designed to support a very specific objective. Similarly, any DSL designed specifically for requirements specification is positioned here.
2. **Platform Independent modeling at lower levels of systems specification using DEVS DSL**: This step involves the development of M2DEVSML or M2DEVS transformations to yield DEVS and/or DEVSML models from CS2F/BM specifications.
3. **Model Structures at higher level of System resolution using Cognitive Domain Ontologies (CDO)**: The CS2F/CDO DSL is founded on the SES theory. This step allows analysis and pruning using the CDOs at higher levels of systems specification and employs model-based repository within the model integrated computing [38] (MIC) paradigm.
4. **Platform Specific Modeling** or DEVS implementations on different platforms: This concept deals with the autogeneration of executable code. The CS2F/BM is executable using DEVSJAVA, or Erlang/OTP. CS2F/CDO is executable using LISP and they are all integrated within the DEVS Netcentric infrastructure.
5. **Platform Specific Modeling i.e. DEVS implementations on different platforms**: This concept deals with the autogeneration of executable code. The CS2F/BM is executable using DEVSJAVA, or Erlang/OTP. CS2F/CDO is executable using LISP. These DSL execution capabilities are all integrated within the DEVS Netcentric infrastructure.
6. **Net-centric execution in a distributed setup**: This concept allows the execution of any DEVSML model in a Netcentric environment where the simulation can be executed in a local-centralized or a remote-distributed setting.

The capabilities defined above allow us to specify any kind of domain models and take the executing real models to live Netcentric systems. A framework for modeling and simulation of the artificial must have these basic capabilities.

3.2 Technical Description of the CS2F Framework

The CS2F framework consists of three components implemented as net-centric services: (1) soaDM, an associative memory based on the declarative memory system

of ACT-R; (2) soaCDO, a domain ontology processing application based on a non-deterministic constraint solver; and (3) the DEVSML Stack, a DEVS-based agent execution framework. Models and agents simulated in the framework base their behavior on: declarative knowledge; cognitive domain ontologies; and behavior models. The CS2F DSLs and framework components used to represent and process these aspects of models and agents are summarized in Table 1.

Table 1 Framework DSLs and the net-centric components in which they are processed

DSL/Formalism	Representation Specialization	Framework Component
CS2F/DM	Declarative Knowledge	soaDM
CS2F/CDO	Domain Knowledge	soaCDO
CS2F/BM	Procedural Knowledge	DEVSML Stack

Declarative, or factual knowledge in a propositional form, is maintained in soaDM. Net services provided by soaDM provide agents with an associative memory through which they can retain and retrieve knowledge. Domain knowledge, or cognitive domain ontologies (CDOs) capturing goals and behavioral design objectives, are maintained in soaCDO. Net services provided by soaCDO provide agents executing in the KCGS framework with an ability to choose *what* to do on the basis of contingencies. Behavior models are predicated non-deterministic finite state machines capturing procedural knowledge in sub-assemblies. Behavior models are maintained and executed in the DEVSML Stack. The DEVSML Stack interacts with the other framework components and realizes the low-level behavior of the agent. Fig. 4 shows the component diagram of CS2F and its Netcentric implementation.

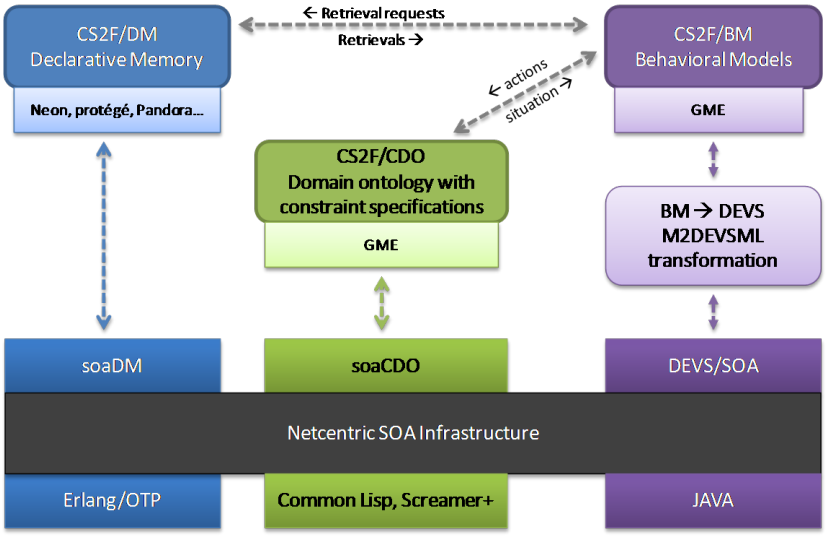

Fig. 4 SOA components in CS2F

3.2.1 CS2F/DM: OWL Ontologies Capturing Declarative Knowledge

Human memory is a part of a quintessential *artificial* system that learns and acts in the world. Human behavior is as flexible as it is because we know lots of things and can use what we know to craft contextually appropriate and effective actions in many different circumstances. Humans know a great deal and can quickly cull through all that they know in order to retrieve and apply the right knowledge given their circumstances. Behavioral flexibility is enabled by a memory system that: (1) provides access to vast amounts of knowledge; and (2) tunes this knowledge to match the information structure of the environment through learning. The ACT-R cognitive architecture [1, 2] includes an associative memory system providing these properties and capabilities. ACT-R's declarative memory system is based on a set of equations explaining the sub-symbolic calculation, learning and utilization of activations and associative strengths [1, 2]. soaDM, a net-centric component of the KCGS framework based on work described by Douglass and Myers [6], utilizes the equations underlying ACT-R's declarative memory system.

Declarative ontologies represent knowledge that models and agents can acquire through experience and retrieve when relevant. CS2F/DM, the DSL with which modelers specify declarative knowledge in soaDM, is based on the OWL ontology standard [17]. CS2F/DM declarative knowledge ontologies describe the classes, class properties, object properties, data properties, and instances constituting a domain. Declarative knowledge ontologies specified in CS2F/DM are translated into files that configure a semantic network in soaDM. Any consistent OWL-compliant ontology can be translated into CS2F/DM and subsequently be used to configure the soaDM semantic network. Because of this, KCGS framework users can take advantage of existing ontologies and RDF databases. Declarative knowledge ontologies can be authored in OWL2-compliant ontology authoring environments such as NeOn, Protégé, Wandora, or Ontopia and then migrated into soaDM. Since these ontology authoring environments support ontology partitioning through namespaces, ontology merging, and knowledge consistency checking, they help KCGS framework users engineer, verify, and understand large-scale declarative knowledge bases.

soaDM is an Erlang/OTP [4] based associative memory through which models and agents can store and retrieve propositional or declarative knowledge. The activation-based associative retrieval mechanism underlying soaDM is based on the declarative module of the ACT-R cognitive architecture [1]. Each node in a semantic network is realized as a separate OTP process thread in Erlang. Activation calculation spreads in soaDM semantic networks as messages are asynchronously exchanged between the process threads constituting their nodes. Since process threads in Erlang execute concurrently, activation-based associative retrieval in soaDM is massively concurrent. See Douglass and Myers [6] for a more comprehensive discussion of how concurrent activation calculation is carried out in soaDM.

3.2.2 CS2F/CDO: Cognitive Domain Ontologies Capturing Contingencies

To be capable of generating autonomous rational action, a model or agent must be able to transform *state descriptions* into *process descriptions*. Transformations of this sort link high-level goals (states) to low-level actions (processes). The vast majority of contemporary cognitive models are built up from productions, rules, or procedural descriptions that combine information about goals and actions. On the surface, this mixing of state and process description knowledge seems to be a natural way of combining the translation of a state description (goal) to a process description (set of actions). A problem with this approach surfaces when it is employed by a modeler specifying a large model that must act autonomously in a complex and dynamic environment: it's almost impossible to specify all the required procedural descriptions combining goals and actions over large spaces of environmental contingencies. In order to express a rich knowledge set that includes environment, contingencies, resources, possible actions and much more, we need a framework that allows us to represent knowledge in many facets or dimensions. The soaCDO is a net-centric component of the framework the uses CS2F/CDO to represent knowledge in such a way.

CS2F/CDO, the DSL in which domain models integrating knowledge related to goals, requirement, situational factors, and possible actions can be specified, is based on System Entity Structure (SES) theory [43]. Cognitive domain ontologies specified in CS2F/CDO are translated into constraint networks in soaCDO. The distinguishing feature between a CDO and an SES representation is the inclusion of constraint language in a CDO. While the SES theory lays the foundation of specifying constraints and how they operate, the CDO constraint language is a formal specification and is an integral part of domain ontology. CS2F/CDO has been developed within the GME, a meta-modeling environment in which domain-specific modeling languages and multi-paradigm modeling frameworks can be formally specified.

The most efficient way to describe CS2F/CDO is to describe its underlying meta-model. Fig. 5 shows the portion of the CS2F/CDO meta-model formally describing correct cognitive domain ontologies (CDOs). The entities, concepts, and relationships constituting the abstract syntax of a DSL are expressed in UML class diagrams in GME. Concepts and entities are represented as classes. Connections terminating with a solid diamond indicate containment relationships between classes. The cardinalities of containment relationships are displayed at their source. Connections terminating with arrows indicate reference relationships. For example, a reference relationship in the Fig. 5 indicates that "NodeReference" entities are allowed to refer to "Node" entities. Triangles denote inheritance; in the figure below a triangle indicates that "Aspect", "Specialization", and "MultiAspect" are all types of "Relationship".

The meta-model in the Fig. 5 specifies that CDOs can contain: nodes/entities; relations, edges/connections; variables; and a variety of references. The meta-model in Fig. 6 shows the portion of the CS2F/CDO meta-model formally describing connections between these elements allowed in valid CDOs. Each allowable connection is represented as a dark circle. The source, destination, cardinality, and associated

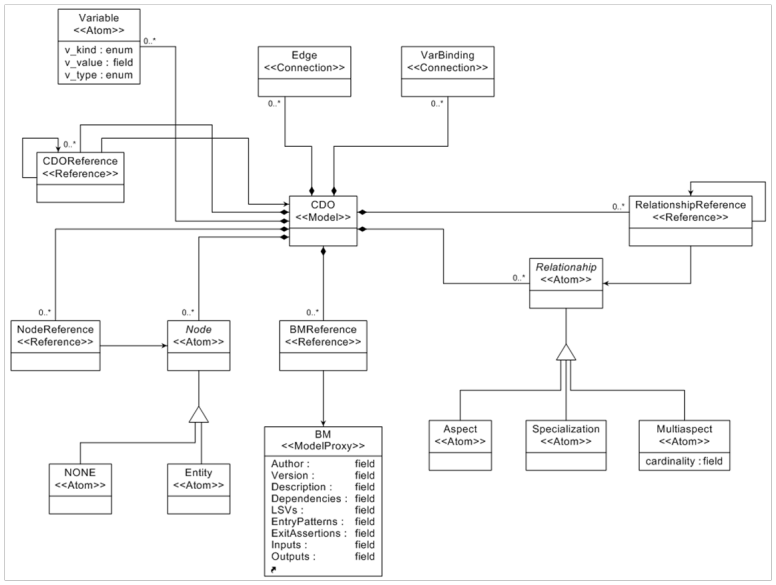

Fig. 5 GME class diagram specifying the portion of the CS2F/CDO meta-model related to valid entities, variables, and relationships in CDO

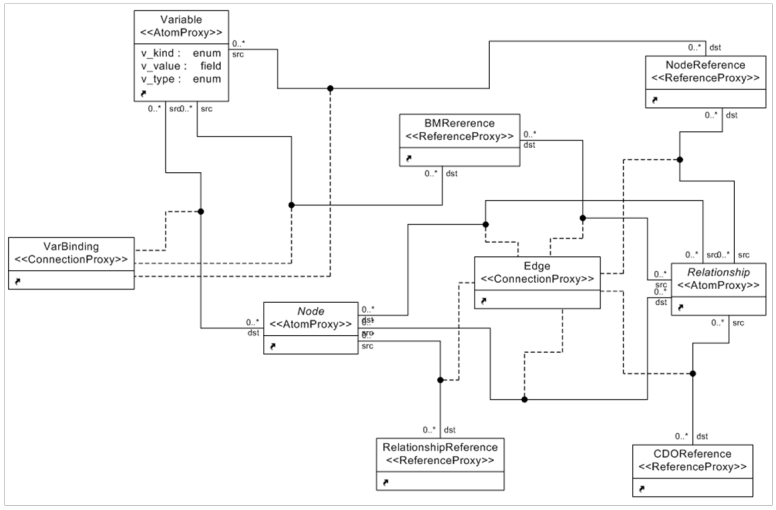

Fig. 6 GME class diagram specifying the portion of the CS2F/CDO meta-model related to valid connections between entities, variables, and relationships in CDOs

connection type constraints in the meta-model work in concert with Object Constraint Language (OCL) constraints (not shown) to enforce axioms underlying SES theory. Entity properties, containment and reference relationships, and constraints in the meta-model ensure that models specified in the CS2F/CDO DSL are “correct

by construction" and therefore do not violate axioms critical in SES theory. GME meta-models can additionally define the concrete syntax or appearance of a DSL. Fig. 7 shows a GME-based CDO authoring environment presenting a graphical/textual concrete syntax for the CS2F/CDO DSL. The DSL's concrete syntax enables CDO authors to combine the following elements in a graphical workspace:

Entity	domain entity (concept) denoted by '**<>**'
Aspect	decomposition (is made up of) denoted by '**|**'
Specialization	can be of type (is a type of) denoted by '**||**'
Multi-Aspect	decomposition into similar type denoted by '**|||**'
Variable	variables attached to entities with ranges or values denoted by '**~**'

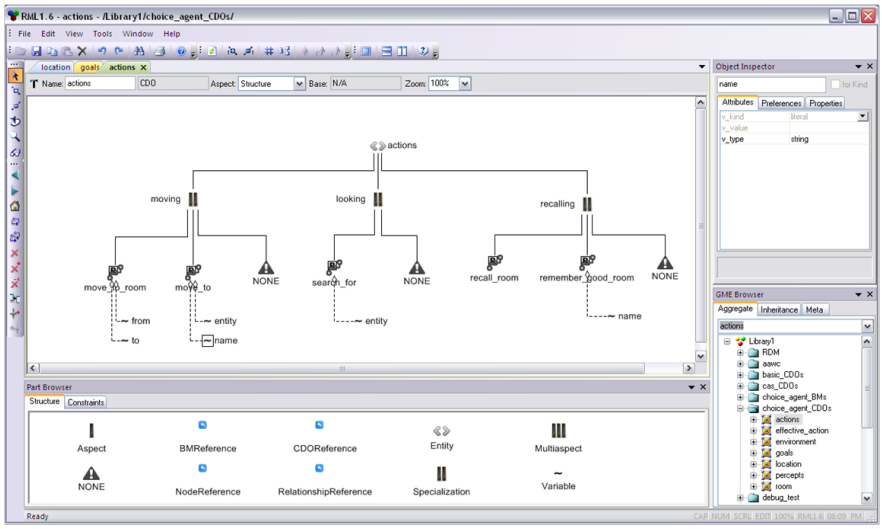

Fig. 7 Cognitive Domain Ontology under development in CS2F/CDO. Note how the GME interface explicitly supports the specification of CDOs

The concrete syntax of CS2F/CDO allows KCGS framework users to graphically specify CDOs containing entities, aspects, specializations, multi-aspects, attached variables, and domain-specific constraints. User actions and choices violating SES axioms during CDO specification are either not allowed by the CS2F/CDO meta-model or generate error messages. This real-time meta-model conformance checking process ensures that CDO are correct by construction.

soaCDO is written in Common Lisp [37]. Non-deterministic programming capabilities based on the Screamer [36] and Screamer+ [40] Common Lisp extensions are used by soaCDO. Screamer adds two basic mechanisms to Common Lisp: (1) a non-deterministic special form called *either* that takes zero or more lisp expressions as arguments; and a deterministic function called *fail* that takes no arguments. The *either* special form non-deterministically evaluates one of the expressions passed

to it, returns the value of the evaluation, and establishes a choice point. The *fail* function triggers back-tracking to the most recent choice point. If un-evaluated expressions are encountered at the choice point, the next value is returned. If no additional expressions are encountered at the choice point, then back-tracking continues to another choice point. Screamer+ extends the functionality of either/fail and allows non-deterministic programming to take advantage of complex data types and Common Lisp Object System [10].

CDOs are computationally realized as structured sets of Common Lisp Object System objects. The root object of a CDO is an instance of a CDO-entity class. CDO-entity instances have a unique name, a collection of zero or more attached CDO-variables, and a collection of zero or more CDO-relations. CDO-variable instances have a unique name and a value. CDO variables can only be connected to CDO entities. Variable instances not assigned a value during initialization have values that Screamer treats as constraint variables. CDO-relation instances have a unique name and a collection of one of more CDO-entities. Aspect, specialization, and multi-aspect classes are derived from the CDO-relation class. Multi-aspect instances have a cardinality. The CDO-entities associated with aspects are maintained in simple lists. The CDO-entities associated with specializations are passed to the *either* special form in order to create a choice point. Establishing specialization entities as a choice point in this way allows Screamer to manage entity enumeration during the computation of constraint system solutions using back-tracking. The CDO-entities associated with multi-aspects are maintained in a list. The number of entities associated with a multi-aspect is a function of the cardinality of the multi-aspect relation.

Table 2 Basic operators in the CS2F/CDO constraint language

Operator	Meaning	Example
`and`	Conjunction	`(and p q)`
`or`	Disjunction	`(or p q)`
`not`	Negation	`(notq)`
`==>`	Implication	`(==> p q)`
`<==>`	Biconditional	`(<==> p q)`
`false`	Logical Falsity	`(and (not p) q)`
`true`	Logical Truth	`(or p (not p))`
`e@`	Entity located in CDO	`(e@ musical_performance style)`
`v@`	Variable attached to CDO entity	`(v@ (actions moving move_to) name)`
`equale`	Entities are equal	`(equale ensemble small-group)`
`equalv`	Variable has a value	`(equalv weight 105)`

`let`	var/val Binding	`(let ((p its-raining)`
		`(q groun-gets-wet)))`
		`(==> p q))`

The CDO pruning process is cast as a constraint-satisfaction problem (CSP) in soaCDO. Constraint variables correspond to CDO-variable values and the entities connected to relations. The domains of constrain variables corresponding to variable values are a function of variable type. CDO-variables can currently be: inte-

gers, floats, strings, lists, vectors, non-numeric enumerated sets, integer ranges, float ranges, and Boolean values. The domains of constraint variables related to a CDO-relation are the set of all entities connected to the relation. Constraints relate sub-sets of the constraint variables and specify the domain values variables are allowed to assume. CS2F/CDO constraints are specified in a language based on first order logic (FOL). Constraints can employ universal and existential quantifiers, implication, bi-directional implication, conjunction, disjunction, negation, and a comprehensive set of non-deterministic functions. Table 2 lists important basic operators that can appear in constraints.

Constraints expressed in well-formed statements are translated into implicative normal form (INF). This translation reduces complex FOL-based statements to disjunctions of implications that are then mapped into Screamer. Appendix 1 lists important complex operators that can appear in constraints that have been translated into INF. The translation into INF produces the following basic implications:

1. Implications consisting of *conventional antecedents* and *conventional consequents*. These are mapped into Screamer as conditional constraints that use *assert!* to propagate constraints in CDOs. For example, the INF implication `(==> p q)` would be mapped into as `(ifv p (assert! q))`
2. Implications with *conventional antecedents* and *consequents equivalent to logical false*. These are mapped into conditional constraints that use fail to trigger back-tracking. For example, the INF implication `(==> p false)` would be mapped into Screamer as `(ifv p (fail))`
3. Implications with *antecedents equivalent to logical true* and *conventional consequents*. These are mapped into unconditional assertions. For example, the INF implication `(==> true q)` would be mapped into Screamer as `(assert! q)`

CDO pruning starts with a process that relates situational factors to corresponding entities in a CDO through the use of the *assert!* operator. These assertions combine with domain-specific constraints in a subsequent search process that finds CSP solutions using a non-deterministic search with chronological back-tracking. The search for CSP solution in soaCDO can: (1) find one solution: (2) find the ith solution: (3) find the "best" solution; (4) find all solutions; or (5) find a solution, present it to the user/agent, and then ask if another solution is required. The ability to obtain solutions from soaCDO while back-tracking over choice points means that CDO with significant combinatory complexity can still be effectively processed.

The example CDO, shown in Fig. 8, is based on an example System Entity Structure discussed in [43]. The CDO represents a set of *musical_performance* entities. Each *musical_performance* has *style* and *ensemble* characteristics; each of which is a specialization. A *musical_performance* can therefore have a style of *symphonic*, *folk*, or *jazz*. A *musical_performance* can also therefore have an ensemble of *orchestra*, *small_group*, or *soloist*. With no additional CS2F/CDO constraints, processing of this CDO in soaCDO would result in 9 CSP solutions generated by crossing all 3 styles with all 3 ensembles. Some of these solutions are clearly implausible. For example, "symphonic soloist" performances are obviously impossible. Implausible

entities such as these can be removed from the set of *musical_performance* entities allowed by the CDO with CS2F/CDO constraints.

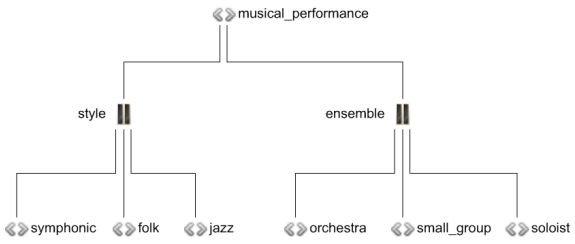

Fig. 8 CDO specifying a space of possible musical performances

As previously mentioned, the CS2F/CDO DSL includes a powerful constraint language that can be used to incorporate domain-specific constraints into CDOs. The translation of these constraints into INF in soaCDO allows a rich constraint language based on FOL to be seamlessly integrated into a CDO processing infrastructure built upon non-deterministic search and chronological backtracking. Table 3 illustrates how CS2F/CDO constraints refining the entity relations in the *musical_performance* CDO translate into INF. Constraints are defined in the CS2F/CDO constrain language using a *define-constraint* macro. The first argument to *define-constraint* is a unique name to be assigned to the constraint. Assigning names to constraints allows KCGS framework users to simplify interactions with soaCDO. The second argument to *define-constraint* is the scope of the constraint. The scope of a constraint is the CDO entity that is to be treated at the root of all entity references in the constraint.

Table 3 Example constraints that refine the possible space of musical performance

Constraint	**Implicative Normal Form**

`(define-constraint m1`
`  :musical-performance`
`  (==> (equale (e@ style)`
`  symphonic)`
`  (equale (e@ ensemble)`
`  orchestra)))`
	`(orv`
	`  (ifv (equale (e@ style) symphonic)`
	`  (assert!`
	`  (equale (e@ ensemble)`
	`  orchestra))))`

`(define-constraint m2`
`  :musical-performance`
`  (==> (equale (e@ style) folk)`
`  (or (equale (e@ ensemble)`
`  small-group)`
`  (equale (e@ ensemble)`
`  soloist))))`
	`(orv`
	`  (ifv (equale (e@ style) folk)`
	`  (assert!`
	`  (orv (equale (e@ ensemble)`
	`  soloist)`
	`  (equale (e@ ensemble)`
	`  small-group)))))`

`(define-constraint m3`
`  :musical-performance`
`  (==> (equale (e@ style) jazz)`
`  (or (equale (e@ ensemble)`
`  small-group)`
`  (equale (e@ ensemble)`
`  orchestra))))`
	`(orv`
	`  (ifv (equale (e@ style) jazz)`
	`  (assert!`
	`  (orv (equale (e@ ensemble)`
	`  orchestra)`
	`  (equale (e@ ensemble)`
	`  small-group)))))`

In Table 3, the constraint *m1* is defined with a scope of *musical-performance* (the root entity in the example CDO shown in Fig. 8). Basing *m1* on this scope allows for the *style* and *ensemble* specializations to be clearly related in a conditional constraint requiring that when the style of the musical-performance is symphonic the ensemble must be orchestra. To ensure that constraints are as computationally efficient as possible, constraint authors should define the scopes of their constraints so that the CSP solution process can "push" constraints as far into the sub-structure of CDOs as possible. The last argument to *define-constraint* is an expression specifying entity/value requirements and variable assignments in the indicated scope. The *m2* and *m3* constraints in Table 3 provide additional domain-specific constraints that refine the domain knowledge captured in the *musical_performance* CDO. An additional constraint limiting the nature of musical performances is provided in Appendix 2. The *m4* constraint in Appendix 2 demonstrates how the transformation to INF allows constraint authors to specify groups of implications in a single constraint. Note how the negations in the consequents of *m4* translate into failure assertions in the implicative normal form. During the CSP solution process in soaCDO, these failure assertions trigger: (1) the elimination of CDO entity/variable assignments; and (2) chronological backtracking.

Table 4 Examples showing how CS2F/CDO constraints impact CSP in soaCDO

Constraints	style	ensemble
none	symphonic	orchestra
	symphonic	small-group
	symphonic	soloist
	folk	orchestra
	folk	small-group
	folk	soloist
	jazz	orchestra
	jazz	small-group
	jazz	soloist
m1, m2, m3	symphonic	orchestra
	folk	small-group
	folk	soloist
	jazz	orchestra
	jazz	small-group
m4	symphonic	orchestra
	folk	small-group
	folk	soloist
	jazz	orchestra
	jazz	small-group

Table 4 lists CSP solutions found by soaCDO under conditions when: (1) no additional domain-constraints were allowed to impact constraint propagation; (2) the simple *m1*, *m2*, and *m3* constraints are allowed to impact constraint propagation; and (3) the complex *m4* constraint defined in Appendix 2 is allowed to impact constraint propagation. Close inspection of the *m4* constraint reveals that it predominately impacts the constraint propagation process through fail-based backtracking. For example, when the ensemble constraint variable is bound to orchestra and the style constraint variable is bound to folk, an assertion of failure immediately eliminates the solution and initiates backtracking.

soaCDO is a Common Lisp based service through which models and agents can represent and process cognitive domain ontologies formally capturing the entities, constraints, and relationships constituting the requirements of the tasks they are performing. soaCDO translates cognitive domain ontologies specified CS2F/CDO into entity/relation networks that are processed with a non-deterministic constraint solver. The constraint-based search/pruning mechanism functions as a type of *cognitive control* allowing models and agents to match their goals to possible actions in such a way that its goals are achieved despite the vagaries of its situation. Cognitive domain ontologies represent knowledge that models and agents are able to process in order to determine ***what*** they should do. Executing in real time, this mechanism allows models and agents to generate behavior *in situ*.

3.2.3 CS2F/BM: Behavior Models Capturing Behavioral Sub-Assemblies

In state-of-the-art cognitive modeling frameworks such as such as ACT-R [2], EPIC [3], and Soar [29], procedural knowledge is specified in productions or rules. Each production is essentially an association between antecedent context requirements and consequent actions. During model simulation, productions whose context requirements are met form a conflict set. Utility calculations or preference are typically used to select which production in the conflict set is allowed to exercise its consequent actions. Unless context is embellished with persistent information, individual productions are unaware of productions that precede or follow them during model simulation. This makes it very difficult to model complex behaviors based on sequences of productions. Modeling frameworks lacking a representation of behavior *above the production* require their users to carefully embellish context with state information if their models depend on behaviors based on sequences of productions. Behaviors based on sequences of productions also must be shielded from interruption. Failure to shield sequences of productions underlying complex behaviors frequently leads to model brittleness in complex dynamic environments.

In the KCGS framework, procedural knowledge is represented in CS2F/BM behavior models; formal structures that allow a modeler to represent behavior above the level of the production [5, 20]. Behavior models can be stored in repositories and used in different contexts. CS2F/BM allows a cognitive modeler to build models and agents from sub-assemblies (behavior models) that conceal complexity rather than large numbers of primitives (productions) that expose complexity. Behavior models are computationally realized as predicated finite state machines. Transitions in behavior models are functionally equivalent to productions; they have pattern-based guards that represent context requirements and side-effects that represent consequent actions. Transition pattern guards are compared to a set of events/facts maintained in a working memory. Behavior models are specified in CS2F/BM, a DSL developed and delivered in the Generic Modeling Environment (GME).

During model execution in the KCGS framework, an agent's behavior is determined by the set of behavior models currently in its *behavior repertoire*. While transition activity in behavior models is typically localized (transitions and generated

actions are concurrent across behavior models), it is possible for them to interact or synchronize through the exchange of events or messages. This allows behavior models to be organized into hierarchies. Discussions of behavior models and their execution can be found in [5, 20].

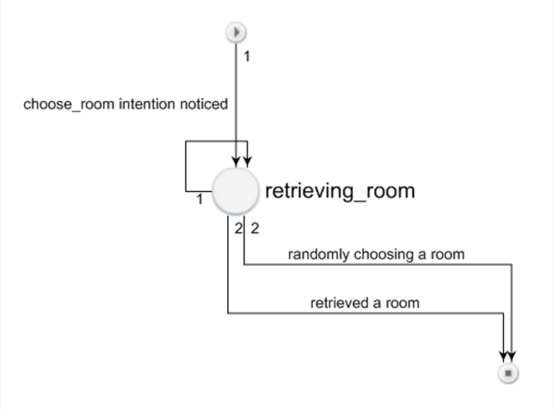

Fig. 9 Graphical representation of a behavior model in CS2F/BM

Fig. 9 presents an example behavior model (BM) in a hybrid graphical/textual concrete syntax. The BM allows an agent to attempt to retrieve a room from soaDM, the associative memory component of the KCGS framework. Transitions in the BM are labeled with brief comments explaining or documenting the purpose of each transition. The state in the BM has been assigned a name that also explains or documents the behavior captured by the BM.

While specifications in the graphical/textual concrete syntax effectively summarize the transitions and state changes underlying a BM, the formal details of the BM remain hidden. The formal details of states and transitions can be specified and edited in GME by selecting a state or transition in an editor and entering attributes in a set of text entry cells. This process is illustrated in [5, 20]. The automated model-to-model translation process that semantically anchors BMs specified in CS2F/BM produces a text-only intermediate description of each behavior model. An example of this textual CS2F/BM form is provided in Table 5.

As illustrated in Table 5, BM transitions can have the following attributes (priority, src, and dst attributes are required):

priority	resolve conflict when more than one transition is possible
label	a description of the function/purpose of a transition.
src	the state from which a transition originates.
dst	the state to which a transition leads.
pre_binds	"name=value" statements used to bind and compare locally scoped variables (LSVs).
patterns	predicate/event constraints that must be met.
functions	execute calculations involving LSVs and context pattern elements.

Table 5 Transition details of the same behavior model specified in a text form generated during the automated transformation of CS2F/BM to executable DEVSML

Behavior Model Transition Details in Textual CS2F/BM

```
transition {
    priority    1
    label       "choose_room intention noticed"
    src         startstate
    dst         retrieving_room
    pre_binds   w=W,context=C
    patterns    {choose_room}
    functions   Endo=[{type, destination}],
                Exo=expand_context(C, W),
                Cs=[]
    assertions  {execute_retrieval, Cs, Endo, Exo}
}
...
transition {
    priority    2
    label       "retrieved a room"
    src         retrieving_room
    dst         stopstate
    patterns    {retrieval_success, C, _},
                {type, C, destination},
                {name, C, CN}
    assertions  {room_chosen, C, CN}
}
```

assertions predicates/events added to working memory after a transition.
post_bindings name/value pairs that overwrite LSVs maintained by a BM.

Predicates/events are represented in transitions as tuples delimited by curly-braces. For example, "{choose_room}" in the patterns of the first transition in Table 5 is a predicate representing an agent intention. In this transition, pre_binds and functions are used to assemble the sub-parts of an assertion that executes a retrieval through soaDM. The second transitions in Table 5 specifies how the sub-parts of a *room_chosen* assertion are to be assembled from properties of a successfully retrieved set of facts about a destination/room.

In the previous section we saw how a DSL such as CS2F/BM can specify behaviors similar to those produced by sets of ACT-R productions. The approach proposed in this section takes the CS2F/BM meta-model in its entirety. The meta-model is semantically anchored in DEVS, which provides solutions to interoperability, extensibility, composability and scalability. CS2F/BM is a recast of our earlier described Research Modeling Language (RML) and detailed transformation is available in [5, 20]. The next subsection provides an overview of the methodology.

From structure perspective, any DEVS system is made up of three elements, the model components (atomic or coupled), the messages that flow between them, and the couplings that communicate these messages between components [46]. Both the atomic and coupled DEVS components transmit and receive messages. However, the capacity to interpret the message and use it to express the behavior is solely the characteristic of a DEVS atomic component. A new message originates exclusively within an atomic component per its behavior specification and is then placed at the output interface of the atomic component. The behavior of an atomic component is a

function of superposition of two behaviors i.e. when an external message is received and when it is not. In order to specify the behavior, a state space is specified and the transitions between these states are defined with respect to an 'event' abstraction.

Describing the richness of DEVS atomic behavior is outside the scope of this paper. We will consider a subset of DEVS formalism known as Finite Deterministic DEVS (FDDEVS) [9]. FDDEVS implemented in the DEVSML 2.0 stack is called the DEVS modeling language [22] that abstracts the DEVS formalism. An automated transformation process using EBNF and Xtext Eclipse Modeling Framework (EMF) is formally specified to preserve the true DEVS semantics. The platform independent DEVS modeling language, as illustrated in Fig. 3 is semantically anchored to the DEVS M&S framework through a middleware.

The notion of 'state' in DEVS is associated with occurrence of an 'event'. Now, looking at each of the transitions in Fig. 9, we find that each transition although specifies the source *src* and the destination *dst* state, has more going on inside it. For example, the *pre_binds*, *post_binds*, *patterns* and *assertions* elements. As per the CS2F/BM semantics, the model will expect the pre_bind variables to match up with the patterns, and if matched, will perform the post_bindings and assertions and will then finally move to the dst state. In DEVS semantics, this operation can be considered as two events, and consequently, two states. The first state being, beginOperation, wherein evaluation is being made per input patterns and the second state being, dst itself. On completion of first state, assertions (output) is being sent and the model then moves to dst state. While there is no problem in the CS2F/BM semantics, the DEVS formalism requires the specification of *output function* which is associated with a specific state. If we preserve the CS2F/BM state set then the point where two events happen together, ie. Incoming patterns and assertions, breaks the notion of discrete event in DEVS formalism. The DEVS semantics very clearly expresses this in the output function. Using the system homomorphism concepts [46] as shown in Fig. 10, by introducing a Zero time state, we not only preserve the CS2F/BM semantics but also transform the state machine into a DEVS state machine. Table 6 lists the mapping of CS2F/BM semantics into FDDEVS elements.

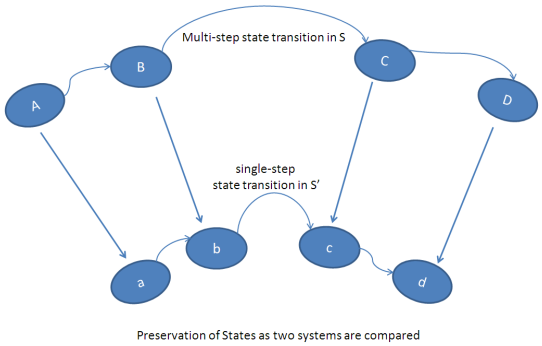

Fig. 10 Preservation of States as two systems are compared and M2DEVS transformation is performed

Table 6 Semantic mapping from CS2F/BM to FDDEVS

CS2F/BM Elements		**FDDEVS Elements**
Globals		
	states	S
Transitions		1. If patterns > 0, then each tuple in patterns is an incoming external message and be addressed in ext. The src state must transition to beginDst state in zero time. 2. If *assertions* > 0 then each tuple is an outgoing message and be addressed in in state beginDst. 3. every beginDst state should internally transition to dst in $0ms$. Every dst must match the $ta = 50ms$ of CS2F/BM state and once elapsed should internally transition to passive.
	src	s in S
	dst	s in S
	patterns	X
	assertions	Y

We have provided an overview on the execution of M2DEVSML transformation from one CS2F/BM DSL into another DSL (DEVSML) that is semantically anchored in DEVS. More details about the atomic behavior, coupling and structure of the transformed CS2F/BM model into DEVS atomic and coupled models can be seen in [20].

4 Modeling in the CS2F Framework

Two agents will be described in the following section. Discussions of how these agents are specified and executed in the KCGS framework illustrate: (1) how declarative knowledge is specified in CS2F/DM (Protégé); (2) how behavior models are specified in CS2F/BM (GME); (3) how cognitive domain ontologies are specified in CS2F/CDO (GME); and (4) how transformed versions of declarative ontologies, behavior models, and CDOs are executed in the net-centric simulation framework. The agents have been simplified so that connections between ontology, epistemology, teleology, and artificial behavior can be clearly and effectively made.

4.1 An Autonomous Agent

Earlier we defined an autonomous agent as one that bases its actions on its contingencies. To be autonomous, such an agent must base its actions on the constraints and affordances of its situation. The first of the agents that will be discussed navigates in a synthetic task environment while searching for a reward item. Through instance-based learning enabled by an associative memory [8], this agent adapts is behavior over time in order to match the information structure of the environment.

This agent represents ***what*** it should do in a CDO, ***how*** it should behave in a set of BMs, and facts it knows and learns in a declarative memory. Rather than basing its behavior on pre-specified rules, the autonomous agent: (1) assigns aspects of its contingencies to entities and variables in a CDO: (2) processes the CDO using a constraint-satisfaction process in order to determine ***what*** it should do: (3) determines ***how*** it should behave by determining which entities in a CSP solution correspond to BMs: and then (4) effectively acts by incorporating these BMs into its behavioral repertoire.

The agent acts in a virtual environment consisting of four rooms. A centrally located room is known as the *home_room*. The other rooms are known as *room1*, *room2*, and *room3*. When the agent moves to a *trigger_plate* in the home room, a reward (small item) randomly appears in *room1*, *room2*, or *room3*. After triggering this event, the agent chooses a room (by retrieving a memory corresponding to it from declarative memory), moves to the room, and then searches the room for the reward. If the reward item is visible in the chosen room, the agent: (1) enters the room: (2) collects the reward; (3) strengthens the activation of the room in declarative memory by making a mental note of the room; and (4) navigates back to the *home_room*. If the reward item is not visible in the chosen room, the agent does nothing. Having collected the reward or not, the agent then returns to the *trigger_plate*.

Initially, the agent has no preference for *room1*, *room2*, or *room3*; the three pieces of declarative knowledge in memory corresponding to the rooms all have the same level of activation. If the appearance of the small item is truly random across the three rooms, then the agent will effectively never come to prefer one room over the others. If the small item is allowed to appear with different probabilities across the rooms, then finding and collecting the item leads to the agent preferring one room over the others. With time and trial repetition, the agent's room preferences adapt to match the reward probabilities of the rooms as mental notes about rooms lead to activation changes in relevant pieces of declarative knowledge.

4.1.1 CS2F/DM – The Agent's Declarative Knowledge

The autonomous agent requires little declarative knowledge to be effective. Appendix 3 summarizes the initial configuration of the agent's declarative memory (maintained by soaDM). Declarative information is represented as nodes in the soaDM semantic network. Edges in the semantic network represent relations between nodes and other nodes (object properties) or numbers/strings (data properties). Properties are always arity/2 (relate 2 things) and have domain and range restrictions. For example, the agent initially knows that *room1*: (1) is of type *destination* (is somewhere is can consider as destination goal); (2) is connected to *home_room*; and (3) has a string name of "room1". An inspection of *door1* reveals that it is both a *way_in* and *way_out* of both *room1* and *home_room*. In other words, nodes and relations represent knowledge that the agent can navigate from *home_room* to *room1* through *door1*.

4.1.2 CS2F/CDO – The Agent's Domain Knowledge

The agent uses a single CDO to determine ***what*** it needs to do in order to achieve its goal of finding and acquiring the small item. The top-level entity in this CDO represents an *effective_action* (or space of behaviors that achieve the design objective of a particular goal). The primary decomposition of *effective_action* is shown in the Fig. 11.

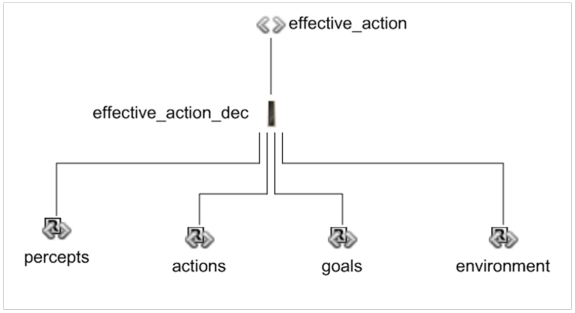

Fig. 11 Top-level entities and relations in the *effective_action* CDO. Note that the percepts, actions, goals, and environment entities are actually reference to previously specified entities in a repository

The CDO formally captures the space of *effective_actions* by decomposing them (through aspect decomposition) to *percepts*, *actions*, *goals*, and the *environment*. This decomposition specifies that *percepts*, *actions*, *goals*, and the *environment* are aspects of *effective_actions*. In the CDO, *percepts*, *actions*, *goals*, and *environment* are actually references to additional CDOs held in a CS2F/CDO repository maintained in GME. References can be expanded when the modeler wishes to view or modify the details of the referred-to entities. The ability to use entity references in CDOs encourages the re-use of CDOs and significantly reduces the visual complexity of large CDOs.

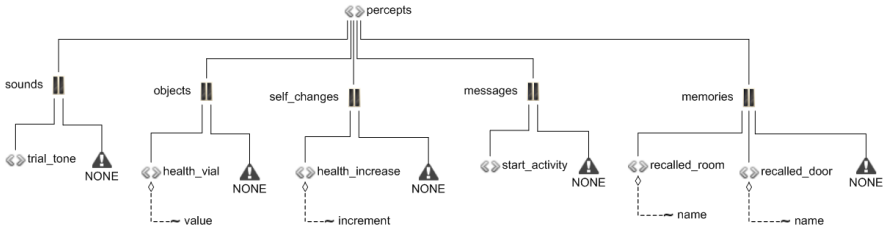

Fig. 12 Component CDO specifying the percepts the autonomous agent can comprehend

Fig. 12 shows that *percepts* entities have characteristics related to *sounds*, *objects*, *self_changes*, *messages*, and *memories*. Each of these characteristics is actually a specialization. The sounds specialization for example, specifies that a percepts sound can either be *trial_tone* or *none*. Instances of the percepts entity correspond

to events/facts the agent is able to perceive. A *health_vial* corresponds to the small item the agent is seeking to locate and acquire. A *health_increase* corresponds to an event/fact generated by the act of collecting the reward item. This percept type is used by the agent to recognize when it has successfully collected the reward item. A *start_activity* corresponds to a message provided to the agent by an operator or experiment frame in order to initiate an agent's behavior. Lastly, the *recalled_room* and *recalled_door* percepts correspond to events/facts retrieved from declarative memory.

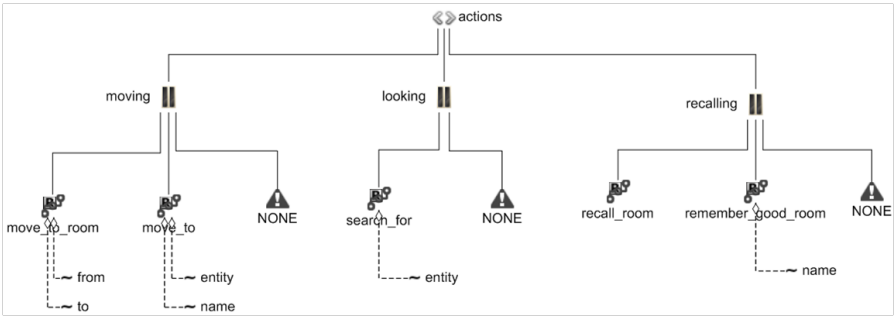

Fig. 13 Component CDO specifying the actions the autonomous agent can initiate

Fig. 13 shows that *actions* entities have characteristics related to *moving*, *looking*, and *recalling*. These characteristics are specializations. The *move_to_room*, *move_to*, *search_for*, *recall_room*, and *remember_good_room* entities in the actions CDO are actually ***references to behavior models***. When the agent processes the *effective_action* CDO *in situ*, instances of these references in CSP solutions will be used to dynamically reconfigure the behavior repertoire of the agent.

Fig. 14 shows that *goals* entities are a combination of a part_task entity expressing the sub-goal underlying an *effective_action* and a desired destination. The destination specialization specifies that goals can involve a desired destinations related to a *room* or *location*. The room entity can be room1, room2, room3, or home_room. The location entity can be door1, door2, door3, trigger_plate, or none.

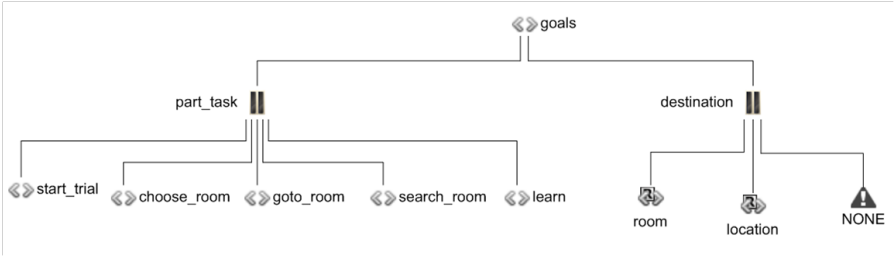

Fig. 14 Component CDO specifying the goals the autonomous agent can maintain

Table 7 illustrates how CDO and domain constraints capture what an agent should do *in situ*. The table shows three constraints that allow the autonomous agent to determine what it should do upon perceiving a *trial_tone* sound in a virtual environment.

Table 7 Constraints allowing the autonomous agent to respond to a sound percept

Examples of the CS2F/CDO Constraint Language

`(define-constraint p_hear_trial_tone`
`  ;; If the trial_tone is heard, then choose a room.`
`  :effective_action`
`  (==> (equale (e@ percepts sounds) trial_tone)`
`  (and (equale (e@ goals part_task) choose_room)`
`  ;; The trial_tone can only be heard in the home_room.`
`  (equale (e@ environment current_room room room_spec) home_room))))`

`(define-constraint c_choose_room`
`  ;; Limits the context in which choose_room is applicable.`
`  :effective_action`
`  (==> (equale (e@ goals part_task) choose_room)`
`  (and (equale (e@ percepts objects) none)`
`  (equale (e@ percepts self_changes) none)`
`  (equale (e@ percepts messages) none)`
`  (equale (e@ percepts memories) none)`
`  (equale (e@ actions looking) none)`
`  (equale (e@ actions moving) none)`
`  (equale (e@ goals destination location location_spec) none))))`

`(define-constraint g_choose_room`
`  ;; To act out choose_room, recall a room from memory.`
`  :effective_action`
`  (==> (equale (e@ goals part_task) choose_room)`
`  (equale (e@ actions recalling) recall_room)))`

Table 8 shows how the CSP solution process provided by soaCDO can use a CDO and additional contingency-based assertions to help an agent determine ***what*** it should do *in situ*. To simplify explication, a direct interaction with the CSP infrastructure of soaCDO is shown. The top part of Table 8 consists of a call to the soaCDO function "soaCDO-solutions". This primitive initiates a non-deterministic CDO search that returns the first configuration of entity and attached variable assignments meeting a CDO's structural constraints and additional domain constraints expressed in the CS2F/CDO constraint language. In Table 8, one CSP solution from the *effective_action* CDO is being requested. The call to soaCDO-solutions includes one additional assertion that maps properties of the agent's contingencies to entities in the *effective_action* CDO. Assertions such as this are essentially function calls accepting two arguments. The first argument is the CDO scope of the assertion. The second argument is the actual assertion. The assertion in Table 8 indicates that in the scope of *percepts* in the *effective_action* CDO, sound is to be constrained to *trial_tone*. This contingent-based assertion and additional CS2F/CDO domain constraints lead to the single CSP solution shown in the bottom part of Table 8. The displayed summary of the CSP solution clearly indicates that under the contingencies expressed by the assertion, the autonomous agent should pursue the action of recalling a room (recall_room).

Table 8 Example showing how CSP in soaCDO results in a CDO solution or prune determining what action(s) the agent should take in order to choose a room

Examples of the CS2F/CDO Constraint Language

```
(soaCDO-solutions
  (effective_action '(assertion :percepts (equale (e@ sounds) trial_tone)))
  :one)

Percepts: sounds/trial_tone, objects/none, self_changes/none, messages/none,
          memories/none
Actions: move/none, look/none, recall/recall_room
Goals: part_task/choose_room, destination/none
Environment: room/home_room, location/none
nil
```

Listings in Tables 7 and 8 illustrate how the autonomous agent acts effectively after perceiving a *trial_tone* sound. The critical thing for the reader to remain aware of is that in the KCGS framework, the agent is determining ***what*** it should do by mapping aspects of its contingencies to a CDO and then using a constraint-satisfaction process to determine how it should use BMs to achieve its goals. The framework allows modelers to exploit an abstraction layer between BMs and CDOs.

The autonomous agent described in this section illustrates how knowledge about contingencies, possible actions, and goals can be represented in CDOs and processed by agents in order to achieve a form of real-time behavioral autonomy. Under these circumstances, CDOs are used to formally relate high-level goals or behavioral design objectives (state descriptions) and low-level actions (process descriptions) in such a way that the agent's behavior is not simply a function of pre-specified rules. CDOs represent connections between contingencies in which agents must act, the agent's goals, and behaviors the agent might utilize to achieve these goals. The key to processing the domain knowledge captured in a CDO under these circumstances is a search process that finds configurations of entities and variables that: (1) meet structural constraints expressed through the aspect, specialization, and multi-aspect relationships in a CDO; and (2) satisfy domain-specific constraints expressed in the CS2F/CDO constraint language.

4.1.3 CS2F/BM – The Agent's Procedural Knowledge

The autonomous agent uses 7 behavior models to generate effective action in the synthetic task environment. The behavior models enable the agent to perform fundamental behaviors necessary for it to act in the task environment. The behavior models are as follows:

1. **assess_separation**: Enables the agent to track the separation distance between itself and a destination location. The separation is reported using the qualitative categories *separated* and *close*.
2. **find_item**: Enables the agent to determine the location of an item by: seeing it directly in percepts; scanning for it in a 90 degree rotation to the left; and scanning for it during a final 180 degree rotation to the right.

3. **move_to**: Enables the agent to either see or recall the location of a named entity and move to it. If the location information is neither visible nor recallable, then the agent rotates until the location information is visible.
4. **move_to_room**: Enables the agent to use a retrieved doorway and the behavior model *move_to* to move an agent from one room to another. If the agent is unable to remember a doorway that leads from its current room to the desired room, the agent initiates a new trial.
5. **recall_room**: Enables the agent to either: retrieve a room that has provided a high frequency of reward items in the past; or randomly choose a room.
6. **remember_good_room**: Enables the agent to increase the activation of declarative knowledge corresponding to a room in which the small item was collected.
7. **search_for**: Enables the agent use the *find_item* and *move_to* behavior models to locate and collect the reward item.

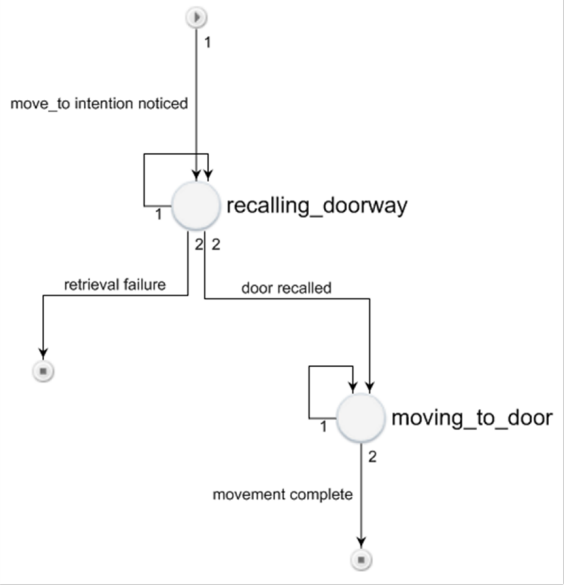

Fig. 15 Graphical representation of a behavior model in CS2F/BM specifying how the autonomous agent can achieve the objective of moving to a room

The *move_to_room* behavior model is shown in Fig. 15. The graphical rendering of the BM shows that the overall behavior involves recalling a door and moving to it. When an agent intends to move to a room, it remembers which door leads to the room and then navigates to the door. When the agent reaches the door, its intended movement is completed and the behavior model transitions to an end state.

Table 9 shows the details of two transitions in the *move_to_room* behavior model. To move to a room, the agent must either visually locate or recall the location of a door that leads from the current room in which it is located and the room it considers its destination. The first of the transitions allows the agent to retrieve a door from declarative memory. Retrieval constraints require that the retrieved door be

Table 9 A partial listing of the transition details of the same behavior model specified in a text form generated during the automated transformation of CS2F/BM to executable DEVSML

Behavior Model Transition Details in Textual CS2F/BM

```
transition {
    priority    1
    label       "move_to intention noticed"
    src         startstate
    dst         recalling_doorway
    pre_binds   context=C,w=W
    patterns    {move_to_room, From, To}
    functions   Constraints=[{type, door}, {way_in, To}, {way_out, From}],
                Endo=[{type, door}],
                Exo=expand_context(C, W)
    assertions  {execute_retrieval, Constraints , Endo, Exo}
    post_binds  from=From,to=To
}
...
transition {
    priority    2
    label       "door recalled"
    src         recalling_doorway
    dst         moving_to_door
    patterns    {retrieval_success, C, _}, {type, C, door}, {name, C, N}
    assertions  {assert_intention, {move_to, C, N}}
    post_binds  door=C,door_name=N
}
...
```

a *way_out* of the room the agent is moving from and an *way_in* to the room the agent is moving to. The second transition allows the agent to actually move to the successfully retrieved door. The transition essentially: (1) verifies that information about a door has been retrieved; (2) obtains the name of the door; and (3) initiates a sub-goal to *move_to* the door. The "{assert_intention, {move_to, C, N}}" assertion in this transition initiates transition activity in the *move_to* behavior model. These two transitions demonstrate how a behavior model can interact with:

- soaDM in order to base behavior on retrieved declarative knowledge
- other behavior models in order to coordinate hierarchical behavior based on the execution of sub-goals

4.1.4 Summary of the Autonomous Agent's Runtime Behavior

When the autonomous agent is initially situated in the simulated task environment, it has: (1) declarative knowledge about rooms, doorways, and the *trigger_plate*. Initially, the agent has a single *central_executive* behavior model in its behavior repertoire. Transitions in the *central_executive* behavior model are sensitive to the following percepts/events:

1. A *start_activity* message originating from a modeler or experiment frame indicating that the agent should begin to perform the overall activity of trying to find and collect the reward object.

2. A *trial_tone* sound originating from the external environment indicating that the agent should initiate a single effort to find the reward object. This sound is produced when the agent stands on the trigger_plate.
3. A *recalled_room* retrieved memory originating from the agent's associative memory indicating which room the agent expects to find the reward object in.
4. A *search_room* goal originating from an internal intention indicating that the agent wants to search for the reward object in a room.
5. A *health_increase* perceived change originating from self-monitoring indicating that the agent has collected the reward object and should make a mental note (increase the activation) of the current room.

As percepts/events originating from outside or inside the agent trigger these transitions, the agent asserts entity and variable values from its contingencies into the *effective_action* CDO and initiates a CSP-based process in soaCDO that "prunes" the CDO. This process utilizes constraints similar to those presented in Table. 7. The agent then integrates any behavior models referred to in one of these CSP solutions into its behavior repertoire. These new, but contextually relevant, behavior models generate effective action until some future percept/event triggers another transition and precipitates another "prune" of the *effective_action* CDO.

The *central_executive* behavior model only transitions when percepts/events require that it re-assess ***what*** it should be doing in order to achieve its goal. Between these transitions, behavior models in the agent's behavior repertoire autonomously tell the story of ***how*** the agent should act in the moment. The process of using cognitive domain ontologies to determine ***what*** to do given contingencies and behavior models to determine ***how*** to act *in situ* allows an agent to translate state descriptions to process descriptions.

4.2 Agents that use an Abduction-Based Inquiry Process

The agent described in the previous section demonstrates how constraint-based processing of CDO can inform an agent ***what*** it should do *in situ*. The agent described in this section demonstrates how CDO can additionally be used by an agent to systematically increase its understanding of its situation. The agent is capable of a type of sensemaking. Before describing this agent, sensemaking and abduction, the type of inference that enables sensemaking, will be defined.

Sensemaking is a process shown in Fig. 16 through which people attempt to understand complex and ambiguous situations so that they can make reasonable decisions and act effectively [12]. In context of this chapter, sensemaking will be defined as abduction-based inquiry.

Abduction can be thought of as a type of inference that plays a role in a process through which inquiry reduces doubt [7, 34]. As a person assesses and understands the context they are trying to act effectively in, they either: (a) find that it's "business as usual" and act according to routine; or (b) are surprised by unexpected events and

observations and try to make sense of things through *designed inquiry*. A surprised person uses abduction, a type of inference from observations to likely explanations or causes, to generate new ideas (hypotheses) about their situation. Through deduction and induction, these hypotheses can be expanded and confirmed/disconfirmed. If necessary, follow-on actions can refine knowledge and hypotheses. The model described in this section uses domain knowledge captured in a CDO and abduction to generate knowledge and hypotheses. This model of abduction is intended demonstrate that an inference-based process (artificial phenomena) that increases the knowledge and autonomy of an agent can be modeled and simulated in the KCGS framework.

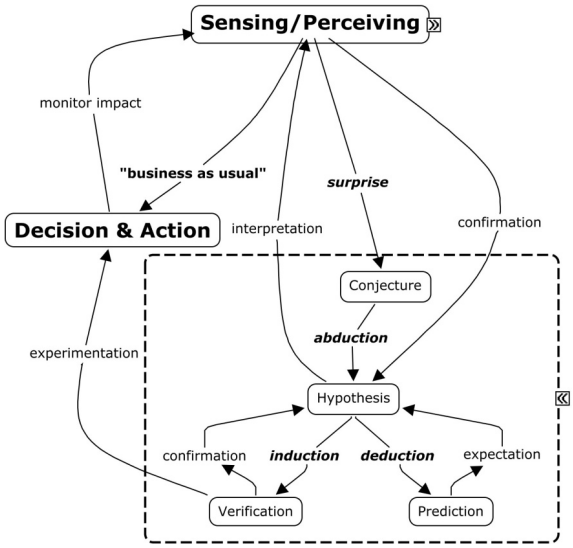

Fig. 16 Central concepts, relations and constraints in a model of sensemaking as abduction-based inquiry

In addition to allowing an agent to determine ***what*** it should do *in situ*, CDO surprisingly allow agents to systematically increase their understanding of situations through an abduction-based inquiry process. The essence of this ability is a non-monotonic reasoning process through which agents: (1) assesses evidence about their situations; (2) assert this evidence and other related aspects of their situations into CDOs representing world knowledge; (3) use constraint propagation to process the CDOs; (4) treat the resulting set of CSP solutions as a hypothesis sets constituting explanations of their situation; and (5) design actions that will allow them to effectively reduce their hypothesis set.

When used this way, CDOs are not just matching contingencies to goals and indicating ***what*** the agent should do *in situ*. Rather, CDOs are being used to capture world knowledge that can be used to relate small-scale observations (evidence) to large-scale ontologies (explanations) in a process that, employing designed action, increases the epistemological quality of the agent's knowledge!

To illustrate this process in as simple an agent as possible, this section will describe an agent that pursues a singular goal of trying to discover the identity of an unknown person. The agent has some general world knowledge about individuals and facial characteristics. The agent knows that certain uniquely identifiable individuals have certain visual characteristics. When asked to guess the identity of an initially unknown person, the agent asks questions designed to constrain the identity of the person so as to systematically refine its knowledge about them.

4.2.1 CS2F/CDO – The Agent's Domain Knowledge

The following figures partially present the CDO used by the identity determination agent. The top-level *guess* CDO in Fig. 17 contains entity references that reduce the visual complexity of the CDO. In order to conserve space, only two of the entity references are presented in full detail in Fig. 18 and Fig. 19. Each entity reference is replaced by the details of the referred to CDO by soaCDO when CS2F/CDO specifications are translated into executable Common Lisp.

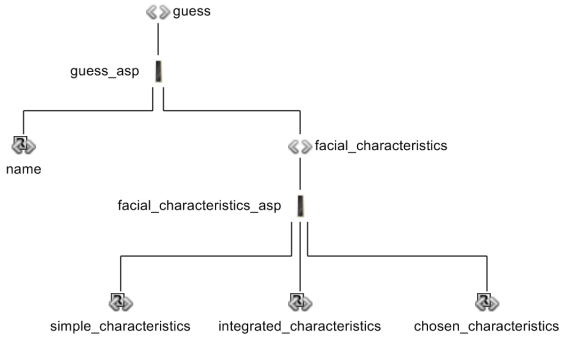

Fig. 17 Top-level entities and relationships in the guess CDO

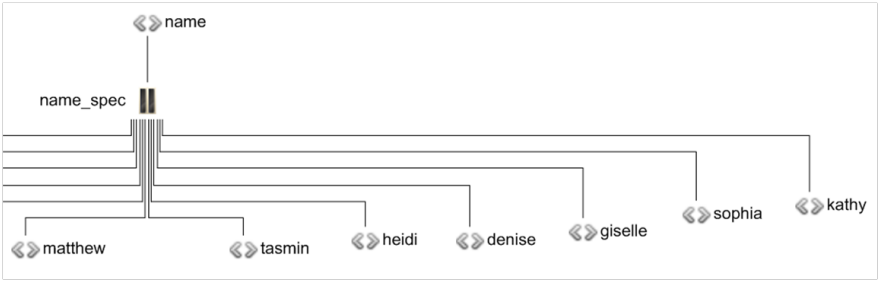

Fig. 18 Component CDO specifying the names guesses can be based on

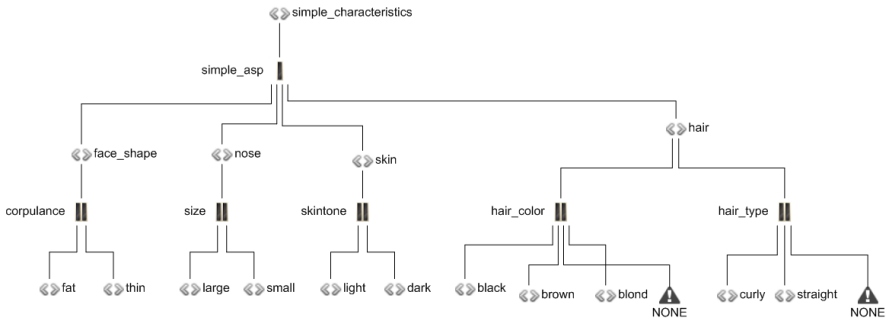

Fig. 19 Component CDO specifying the simple_characteristics guesses can be based on

The *guess* CDO specifies a space of identities. Without additional domain constraints, this CDO would produce a large number of solutions or prunes when processed by soaCDO. These solutions would combine names with constellations of simple, complex, and chosen characteristics. Without domain constraints specifying the unique characteristics of each guess name, multiple solutions based on each name would exist.

If constraints similar to those shown in Table. 10 are defined and incorporated into the CDO, then each *guess name* is constrained to have a specific set of characteristics. Under these circumstances, the *guess* CDO captures a set of named identities with fixed characteristics. With these domain constraints, only one solution for each *guess name* can exist.

Table 10 Example constraints refining the aspects of guesses. Note how these constraints "define" individuals by relating a set of characteristics to the *name* of a *guess*

Examples of the CS2F/CDO Constraint Language

```
(define-constraint adam
  :guess
  (let ((<simple_asp> (n@ guess ... simple_asp))
        (<integrated_asp> (n@ guess ... integrated_asp))
        (<chosen_asp> (n@ guess ... chosen_asp)))
    (==> (equale (e@ guess guess_asp name name_spec) adam)
         (and (equale (e@ <simple_asp> face_shape corpulance) fat)
              (equale (e@ <simple_asp> nose size) small)
              (equale (e@ <simple_asp> skin skintone) light)
              (equale (e@ <simple_asp> hair hair_color) brown)
              (equale (e@ <simple_asp> hair hair_type) straight)
              ;;
              (equale (e@ <integrated_asp> countenance countenance_spec) smile)
              (equale (e@ <integrated_asp> gender_spec) male)
              (equale (e@ <integrated_asp> age age_spec) old)
              ;;
              (equale (e@ <chosen_asp> facial_hair mustache_spec) none)
              (equale (e@ <chosen_asp> facial_hair beard_spec) none)
              (equale (e@ <chosen_asp> headgear headgear_spec) hat)
              (equale (e@ <chosen_asp> eyeware eyeware_spec) none)))))
... Additional Constraints ...
```

Possessing world knowledge capturing information about the characteristics of named individuals, the identity determination agent is able to guess the unknown individual by systematically acquiring knowledge about his or her characteristics. To acquire knowledge about the characteristics of the unknown individual, the agent simply asks if they have a specific characteristic. Each answer to these queries becomes an assertion that reduces the number of subsequent CSP solutions found by soaCDO. If CDO solutions are considered hypotheses about the identity and characteristics of the unknown individual, then each assertion reduces the number of hypotheses. This process clearly uses inquiry to reduce doubt. The listing in Table. 11 shows how 4 assertions reduce the hypothesis space represented in the *guess* CDO to 2. The assertions indicate that previous questions led to knowledge that the individual: (1) is wearing a hat; (2) is not wearing glasses; (3) is female; and (4) is young. When these characteristics are asserted as requirements during the constraint propagation process, only 2 solutions are found. To continue to make sense of the identity of the unknown individual, the agent would note that the remaining hypotheses differ with respect to *hair_color* and ask "Does the unknown person have black hair?" The answer to this query would provide the last piece of evidence required to disambiguate the identity of the unknown individual.

Table 11 Example showing how CSP in soaCDO results in a set of CDO solutions constituting the hypothesis space resulting from abducing from evidence of characteristics to explanations based on named guesses meeting constraints based on the evidence

Example CSP Solution

```
(soaCDO-solutions
  (guess_ '(assertion :integrated_characteristics (equale (e@ age_spec) young))
          '(assertion :integrated_characteristics
                      (equale (e@ gender_spec) female))
          '(assertion :chosen_characteristics
                      (notv (equale (e@ eyeware_spec) glasses)))
          '(assertion :chosen_characteristics (equale (e@ headgear_spec) hat)))
  :print)

Name: sophia
Simple Aspects: face_shape/thin, nose/small, skintone/light, hair_color/blond,
                hair_type/curly
Integrated Aspects: expression/smile, gender/female, age/young
Chosen Aspects: facial_hair/(none, none), headgear/hat, eyeware/none

Do you want another solution? (y or n) >> y

Name: petra
Simple Aspects: face_shape/thin, nose/small, skintone/light, hair_color/black,
                hair_type/curly
Integrated Aspects: expression/smile, gender/female, age/young
Chosen Aspects: facial_hair/(none, none), headgear/hat, eyeware/none

Do you want another solution? (y or n) >> y
nil
```

The ambiguous situation the identity determination agent is trying to make sense of centers on the ambiguous identity of an individual. When the agent is initially asked to make sense of its situation, it is unable to discount any of the named indi-

viduals it has knowledge about in the *guess* CDO. The hypothesis space the agent seeks to reduce using an abduction-based inquiry process contains all named individuals. The agent reduces the hypothesis space by asking questions about characteristics distinguishing a subset of the hypothesis space. The agent uses a CDO and the assertion of accumulating evidence to refine its knowledge of the identity of the unknown individual. Using a CDO in this way is quite different than using one to determine what to do *in situ* since it enables an agent to refine its knowledge.

5 The KCGS Framework

In order to express a rich knowledge set that includes environment, contingencies, resources, possible actions and much more, we need a framework that allows us to represent knowledge in many facets or dimensions. While a cognitive rational agent uses all this knowledge to compose its immediate action, it is very difficult as a modeler to construct this knowledge-set if there is only one dimension. For a multi-dimensional and multi-resolutional knowledge representation, the knowledge framework must itself allow constructions of this kind of representation. Ontology, in technical terms is a graph of nodes and information is presented in the relations that exist between these nodes. Of course, it is a great step as the knowledge can now be presented in associative terms, more like a semantic network. It is now more amenable to data engineering efforts but it is essentially flat and not suitable for piecewise construction or layered methodologies for better manageability. The SES formal knowledge representation mechanism with its set of axioms and rules helps develop an ontology that can be constructed and deconstructed in piecewise manner through SES aspects and specializations. The latest work in SES ontology domain is an evidence of such efforts [43].

We have shown in our narrative earlier how an agent can have its description in multiple aspects and specializations. Such aspects and specialization can be added or removed incrementally and intuitively without changing other facets of the system and still understandable by the common modeler. In other words, the modeler is not overwhelmed by the influx of new knowledge as it builds upon the existing ones. This is important because in large systems, large knowledge-set often results in 'information paralysis' at the modeler end. Such aspects and specializations give ontology a multi-resolutional capability and can be called upon at real-time execution of the system. Also note here that adding such elements is piecewise isolated and it is the defined rules that create relationships between different SES elements at run-time thus managing complexity. It also implies that while the general structure of the proposed ontology remains intact, it is the defined rules that dictate the association and affordances of the entire system at run-time. These rules then become dynamic and dictate how the knowledge entities interact. This property of SES is a major way forward as compared to existing cognitive models where the rules are a function of the invariant architecture itself and any change in the architecture calls for major upgrades in the modeling system. The realization of these rules by DEVS

formalism in a SES modeled system is much easier, manageable and formally verifiable at run-time.

Another advantage of this piecewise construction is partitioning of the expert knowledge in the domain of interest. It now becomes much more feasible to integrate the expert knowledge of other cognitive scientists as aspects of such ontology. Therefore, we attempt to construct and open the proposed Cognitive Domain Ontology for further input and contributions from the community at-large. Once the structure of these aspects is laid out, it is easier to define and modify rules that related different aspects of the ontology.

5.1 Putting CS2F in the KCGS perspective

This chapter describes the CS2F framework in order to illustrate how *artificial systems* producing *artificial phenomena* can be modeled and simulated. The framework combines aspects of SES theory, DEVS-based general systems theory, cognitive architectures (ACT-R), and DSL development using meta-modeling to change the way artificial systems are formally specified and simulated. The presentation of the CS2F framework and example agents has been tailored to the objective of highlighting how ontology, epistemology, and teleology play roles in the realization of autonomous models and agents in the framework. The framework is significantly more than just a set of net-centric applications capable of executing a set of new DSLs though. This section will describe how CS2F is an instance of a larger KCGS framework.

The KCGS framework is based on three major areas with formal SES theory at their centers:

I Ontology and data representation.
II Knowledge engineering and parallel distributed computation search mechanisms.
III DEVS Unified Process.

I deals with knowledge representation and how data interoperability is achieved between different ontologies using SES foundational framework. In its current state, basic programmatic pruning mechanisms are used. **II** deals with the entire knowledge engineering and data-mining aspect of executing the pruning process that transform data into information. This computational process has to align with the AI-based search mechanisms, and real-time execution capabilities that will lead to formal SES-based pruned SESs. Finally, **III** takes the formal PESs and using the DEVS M&S technology, provides the requirement traceability, platform independent M&S, Verification and Validation and various other capabilities such as SOA execution, and system component descriptions in DEVS Unified Process.

The capabilities of the KCGS framework as realized in CS2F allow us to specify many kinds of domain models and take the executing real models to live netcentric systems. A framework for modeling and simulation of the artificial must have these

basic capabilities to incorporate large-scale heterogeneous systems. Table 12 lists some of the requirements of such a framework and Fig. 20 shows how CS2F and the larger KCGS framework address these requirements.

Table 12 Mapping requirements for M&S Framework for the Artificial with CS2F and KCGS components

Framework Requirements	Technical Foundation	CS2F Component	KCGS Component
Based on General Systems Theory	DEVS System Theory		DEVS Unified Process
Facilitate model-based development and engineering	DEVS M&S Framework, SES Theory	CDO	SES Theory
Scalable and component-based	DEVS M&S Framework	BM, CDO	DEVSML 2.0, DEVS Unified Process
Manage Hierarchy and abstractions	DEVS Systems Theory	BM, CDO	DEVS Unified Process
Interoperable across implementation platforms	DEVS M&S Framework		DEVSML 2.0
Formal specification	DEVS M&S Framework	BM, CDO, DM	DEVS Unified Process, SES Theory
Domain and platform neutral	DEVS Systems Theory, SES Theory	CDO	Ontology and Knowledge Representation
Agile and persistent	DEVS Systems Theory, SES Theory	CDO, DM, BM	DEVS Unified Process, SES Theory
Interface with AI knowledge engineering methodologies	SES Pruning	CDO Pruning	SES Theory, Data Mining, Constraint Satisfaction Problem (CSP)

Fig. 20 also shows how different disciplines interact together and interface with the formal SES theoretical framework. CS2F/DM is a DSL that formally captures declarative knowledge in ontologies. CS2F/BM interfaces with DEVSML 2.0 stack through various transformations. CS2F/CDO works at the intersection of SES theory, constraint satisfaction problems and various other knowledge engineering measures as overlaid on SES theoretical framework. The DEVS Unified Process is a superset that incorporates formal DEVS System theory, platform transparent M&S layered framework called as DEVSML 2.0, requirements engineering at the intersection with formal domain ontology representations using SES and other methods.

Ultimately, the solution we are looking for is an ontological framework that lends itself seamlessly into the simulation-based component modeling framework. Fig. 21 presents the meta-meta-model of an autonomous system. It formally captures real-world facets like environment and resources and agent-based facets like goals and behavior. Constraints play a dual role in an Autonomous System's ontology [19]. There are two types of constraints. Type I constraints are physics based (hard truths) and Type II constraints are situation-based. While Type I constraints are hard constraints, the Type II constraints are soft constraints that are dynamic. The Type II

constraints are the ones that are responsible for contingency based behavior and situated behavior. The pruning process will work on these Type II constraints to generate a CDO that is 'situated'.

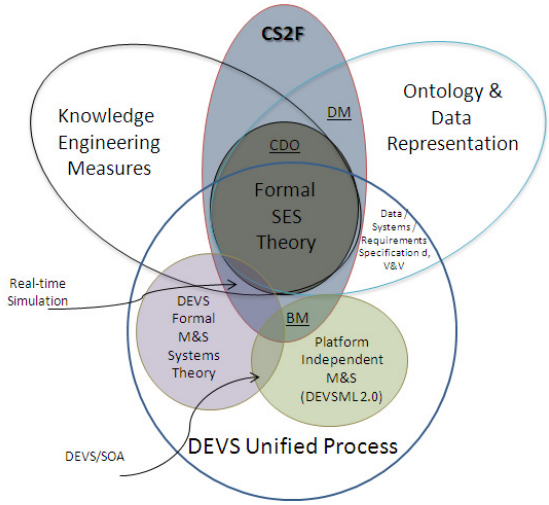

Fig. 20 Putting CS2F in perspective of Knowledge-based Contingency-driven Generative Sytems framework

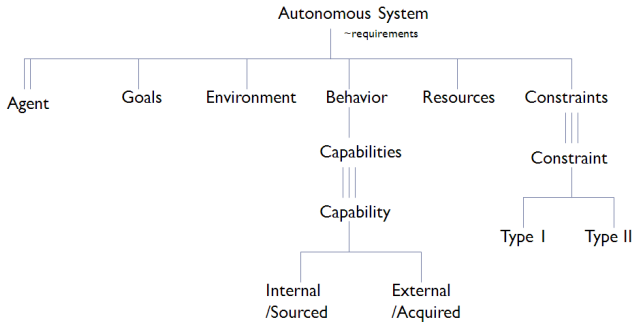

Fig. 21 Meta-meta-model for an Autonomous System

Earlier research demonstrated that SES, rule-based search processes, and conventional simulation can be used to capture, search through, and evaluate system configuration spaces. These efforts demonstrated how a rule-based search or SES pruning process can derive system configurations that meet the design objectives explicit in the SES. Current research efforts have demonstrated that SES can capture domains other than physical system design spaces. The KCGS framework described in this chapter combines current and previous patterns of SES use in modeling and simulation by: (1) using CDOs to represent behavior configuration spaces consist-

ing of agents (beliefs, goals, behavioral constraints) and situational contingencies (task requirements, action affordances, physical constraints); (2) searching through (pruning) CDOs at runtime in order to generate behaviors that meet the goals and contingencies; and (3) executing cognitive agents employing CDOs in larger M&S framework such as DEVS Unified Process [18]. Future work will refine the KCGS framework and explore the relationships between declarative, procedural, and domain knowledge in a formal modeling and simulation framework founded on the DEVS [46] Unified Process [18, 27]. Future work will also explore how large-scale autonomous (artificial) models and agents can be integrated into systems of systems and Human-in-loop solutions.

6 Concluding Remarks

The cognitive system specification framework (CS2F) is a composition of DSLs tailored to the needs of cognitive modelers. The abstract syntaxes of two of the DSLs composed in CS2F (CS2F/DM and CS2F/BM) are strongly influenced by the ACT-R cognitive architecture [1,2]. The abstract syntax of CS2F/CDO is influenced by System Entity Structure (SES) theory [43]. The concrete syntaxes of CS2F/DM and CS2F/BM are designed so that a modeler with experience in ACT-R can specify behaviorally equivalent models at a high level of abstraction. The concrete syntax of CS2F/CDO is designed to allow modelers to rapidly specify theoretically sound CDOs regardless of their experience working with SES.

This chapter explored how a new modeling and simulation framework can allow a modeler to: (1) formally represent actions as behavior models in CS2F/BM; (2) formally represent goals and contingencies as cognitive domain ontologies in CS2F/CDO; and (3) let autonomous models and agents decide ***what*** to do on their own *in situ*. The framework consists of three major net-centric components each representing and processing a unique type of agent knowledge:

soaDM an Erlang/OTP based associative memory through which models and agents can store and retrieve propositional or declarative knowledge.

soaCDO a Common Lisp based service through which models and agents can represent and process cognitive domain ontologies formally capturing the entities, constraints, and relationships constituting the requirements of the tasks they are performing.

DEVSML Stack a DEVS/Java based integration service through which models and agents can represent and process behavior models formally capturing actions they can perform.

Models and agents technically realized in the KCGS framework do not use predefined rules and knowledge that interleave state and process descriptions to act in anticipated circumstances. Instead, they are persistent computational entities that use CDO search/pruning *in situ* to re-configure their own behavioral repertoires

to match their objectives and goals to their contingencies. The generative framework allows modelers to specify behavior above the level of the CS2F/BM behavior model. The 3 components of the generative framework discussed above are computationally realized in the modeling and simulation infrastructure developing in the LSCM initiative [21]. The DEVSML Stack translates behavior models specified in CS2F/BM into DEVS coupled models which are then executed in a net-centric realization of DEVS [20].The generative framework represents a significant modeling capability advancement that will add to, and leverage, the DSLs and M&S capabilities being researched and developed in the LSCM initiative.

6.1 Roles of Ontology, Epistemology, and Teleology in Artificial Systems

The KCGS framework discussed in this chapter supports the modeling and simulation of autonomous agents. These agents base their behaviors on their contingencies not just pre-specified rules. Autonomous agents modeled in the KCGS framework use three knowledge representations to gain autonomy: (1) procedural knowledge; (2) declarative knowledge; and (3) domain knowledge. These three knowledge representations allow KCGS framework users to model and simulate agents that exploit ontologies, produce artificial behaviors that are teleological, and refine their knowledge over time. The framework's effectiveness can be attributed to its exploitation of:

Ontology Declarative and domain knowledge are represented using ontologies. Declarative knowledge is described in OWL ontologies, translated into semantic networks, and processed by agents in a soaDM associative memory component of the framework. Domain knowledge is described in CDOs, translated into constraint networks, and processed by agents in a soaCDO component.

Teleology Agents use CDOs to determine ***what*** they should do *in situ*. Agents do this by mapping characteristics of their contingencies into CDOs and propagating them through: (1) constraints reflecting the aspects, specializations, and multi-aspects characterizing the CDOs; (2) domain constraints specified in a CS2F/CDO constraint language. When CDOs represent spaces of behaviors that achieve a behavioral design objective (goal), they allow an agent to generate goal-pursuing behavior in complex and dynamic environments.

Epistemology Agents use CDOs to infer abductively from observed evidence to likely explanations. Under these circumstances, agents relate observations (in the form of asserted evidence) to what they know about the world (in the form of a CDO). By selecting actions that elicit additional evidence from the environment, agents can refine their situational knowledge.

Appendix 1: Complex operators in the CS2F/CDO constraint language

Operator	Meaning	Example
assert!	Set value of a constraint variable	(assert! (equalv (v@ (weight) kg) 100))
andv	Conjunction with variables	(andv (equale (e@ aspect sport) golf) (equale (e@ aspect size) small))
orv	Disjunction with variables	(orv (equale (e@ aspect size) small)) (equale (e@ aspect size) large))
notv	Negation with variables	(notv (equale (e@ ensemble) soloist))
ifv	Implication with variables	(ifv (equale (e@ ensemble) soloist) (notv (equale (e@ style) symphonic)))
fail	Failure with backtracking	(ifv (andv (equale (e@ ensemble) soloist) (equale (e@ style) symphonic)) (assert! (fail)))
a-member-of	Non-deterministic selection from a list [creates choice point]	(assert! v (a-member-of '(a s d f)))
either	Non-deterministic selection from arguments [creates choice point]	(either small medium large)
an-integer-above an-integer-between an-integer-below	Define integer ranges	(equalv v (an-integer-above 10)) (assert! (equalv (v@ (weight) kg) (an-integer-between 75 105)))
a-real-above a-real-between a-real-below	Define real ranges	(equalv pi (a-real-between 3.0 4.0)) (ifv (notv (equalv pi 3.141592653589793)) (fail))
>v, >=v, <v, <=v	Comparison functions that accept constraint variables	(ifv (<=v v 10) (fail))

Appendix 2: Example constraints that refine the possible space of musical performance

Constraint	Implicative Normal Form
(define-constraint m4 :musical-performance (and (==> (equale (e@ style) jazz) (or (equale (e@ ensemble) small-group) (equale (e@ ensemble) orchestra))) (==> (equale (e@ style) folk) (not (equale (e@ ensemble) orchestra))) (==> (equale (e@ style) symphonic) (not (or (equale (e@ ensemble) soloist) (equale (e@ ensemble) small-group))))))	(orv (ifv (equale (e@ style) jazz) (assert! (orv (equale (e@ ensemble) orchestra) (equale (e@ ensemble) small-group)))) (ifv (andv (equale (e@ ensemble) orchestra) (equale (e@ style) folk)) (assert! (fail))) (ifv (andv (equale (e@ ensemble) soloist) (equale (e@ style) symphonic)) (assert! (fail))) (ifv (andv (equale (e@ ensemble) small-group) (equale (e@ style) symphonic)) (assert! (fail))))

Appendix 3: Declarative knowledge available to the autonomous agent through soaDM

SemNet Node	Object Properties	Data Properties
home_room	type = origin connected_to = room1 connected_to = room2 connected_to = room3	name = "home_room"
room1	type = destination connected_to = home_room	name = "room1"
room2	type = destination connected_to = home_room	name = "room2"
room3	type = destination connected_to = home_room	name = "room3"
door1	type = door way_in = room1 way_in = home_room way_out = home_room way_out = room1	name = "door1"
door2	type = door way_in = room2 way_in = home_room way_out = home_room way_out = room2	name = "door2"
door3	type = door way_in = room3 way_in = home_room way_out = home_room way_out = room3	name = "door3"
trigger_platec	trigger_platec	name = "trigger_plate" location_x = 3653.0 location_y = 1975.0 location_z = -197.65

References

1. Anderson, J.: How can the human mind occur in the physical universe?, vol. 3. Oxford University Press, USA (2007)
2. Anderson, J., Bothell, D., Byrne, M., Douglass, S., Lebiere, C., Qin, Y.: An integrated theory of the mind. Psychological Review **111**(4), 1036 (2004)
3. Anderson, J., Matessa, M.: An overview of the epic architecture for cognition and performance with application to human-computer interaction. Human-computer interaction **12**(4), 391–438 (1997)
4. Cesarini, F., Thompson, S.: Erlang programming. O'Reilly Media (2009)
5. Douglass, S., Mittal, S.: Using domain specific languages to improve scale and integration of cognitive models. In: Proceedings of the Behavior Representation in Modeling and Simulation Conference. Utah, USA (2011)
6. Douglass, S., Myers, C.: Concurrent knowledge activation calculation in large declarative memories. In: Proceedings of the 10th International Conference on Cognitive Modeling, pp. 55–60 (2010)
7. Fann, K.: Peirce's theory of abduction. Martinus Nijhoff La Haya (1970)
8. Gonzalez, C., Lerch, J.F., Lebiere, C.: Instance-based learning in dynamic decision making. Cognitive Science **27**(4), 591–635 (2003)
9. Hwang, M., Zeigler, B.: Reachability graph of Finite and Deterministic DEVS networks. IEEE Transactions on Automation Science and Engineering **6**(3), 468–478 (2009)
10. Keene, S.: Object-oriented programming in Common Lisp: A programmers guide to CLOS. Adison-Wesley (1989)
11. Kim, T., Lee, C., Christensen, E., Zeigler, B.: System entity structuring and model base management. Systems, Man and Cybernetics, IEEE Transactions on **20**(5), 1013–1024 (1990)
12. Klein, G., Phillips, J., Rail, E., Peluso, D.: A data-frame theory of sensemaking. In: Expertise out of context: proceedings of the sixth International Conference on Naturalistic Decision Making, p. 113. Lawrence Erlbaum (2007)
13. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing, Budapest, Hungary, vol. 17 (2001)
14. Ledeczi, A., Volgyesi, P., Karsai, G.: Metamodel composition in the Generic Modeling Environment. In: Comm. at workshop on Adaptive Object-Models and Metamodeling Techniques, Ecoop, vol. 1 (2001)
15. Lee, H., Zeigler, B.: SES-based ontological process for high level information fusion. In: Proceedings of the 2010 Spring Simulation Multiconference, p. 129. ACM (2010)
16. Lee, H., Zeigler, B.: System entity structure ontological data fusion process integrated with C2 systems. The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology **7**(4), 206–225 (2010)
17. McGuinness, D., Van Harmelen, F., et al.: OWL web ontology language overview. W3C recommendation **10**, 2004–03 (2004)
18. Mittal, S.: DEVS Unified Process for integrated development and testing of Service Oriented Architectures. Ph.D. thesis, Iniversity of Arizona (2007)
19. Mittal, S.: Net-centric cognitive architecture using DEVS Unified Process. In: Researching and Developing Persistent and Generative Cognitive Models Workshop. Scottsdale, AZ (2010)
20. Mittal, S., Douglass, S.: From domain specific languages to DEVS components: application to cognitive m&s. In: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, pp. 256–265. Society for Computer Simulation International (2011)
21. Mittal, S., Douglass, S.: Net-centric ACT-R-based cognitive architecture with DEVS Unified Process. In: Proceedings of the 2011 Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium, pp. 34–44. Society for Computer Simulation International (2011)

22. Mittal, S., Douglass, S.: DEVSML 2.0: The language and the stack. In: In Proceedings of the Spring Simulation 2012 Multiconference. Orlando, FL (2012)
23. Mittal, S., Risco-Martin, J.: Netcentric System of Systems Engineering with DEVS Unified Process. CRC Press (2012)
24. Mittal, S., Risco-Martin, J., Zeigler, B.: DEVS-based simulation web services for net-centric T&E. In: Proceedings of the 2007 summer computer simulation conference, pp. 357–366. Society for Computer Simulation International (2007)
25. Mittal, S., Risco-Martín, J., Zeigler, B.: DEVSML: automating DEVS execution over SOA towards transparent simulators. In: Proceedings of the 2007 spring simulation multiconference-Volume 2, pp. 287–295. Society for Computer Simulation International (2007)
26. Mittal, S., Risco-Martín, J., Zeigler, B.: DEVS/SOA: A cross-platform framework for net-centric modeling and simulation in DEVS Unified Process. Simulation **85**(7), 419–450 (2009)
27. Mittal, S., Zeigler, B., Risco-Martin, J.: Implementation of formal standard for interoperability in M&S/systems of systems integration with DEVS/SOA. International Journal of Command and Control **2** (2009)
28. Molnár, Z., Balasubramanian, D., Lédeczi, A.: An introduction to the Generic Modeling Environment. In: Proceedings of the TOOLS Europe 2007 Workshop on Model-Driven Development Tool Implementers Forum. Zurich, Switzerland (2007)
29. Newell, A.: Unified theories of cognition, vol. 187. Harvard Univ Pr (1994)
30. Risco-Martín, J., Moreno, A., Cruz, J., Aranda, J.: Interoperability between DEVS and non-DEVS models using DEVS/SOA. In: Proceedings of the 2009 Spring Simulation Multiconference on ZZZ, p. 147. Society for Computer Simulation International (2009)
31. Rozenblit, J., Hu, J., Kim, T., Zeigler, B.: Knowledge-based design and simulation environment (KBDSE): Foundational concepts and implementation. Journal of the Operational Research Society pp. 475–489 (1990)
32. Rozenblit, J., Huang, Y.: Rule-based generation of model structures in multifaceted modeling and system design. ORSA Journal on Computing **3**(4), 330–344 (1991)
33. Rozenblit, J., Zeigler, B.: Representing and constructing system specifications using the system entity structure concepts. In: Proceedings of the 25th conference on Winter simulation, pp. 604–611. ACM (1993)
34. Schvaneveldt, R., Cohen, T.: Abductive reasoning and similarity: Some computational tools. Computer-Based Diagnostics and Systematic Analysis of Knowledge pp. 189–211 (2010)
35. Simon, H.: The sciences of the artificial, 2nd edn. the MIT Press (1981)
36. Siskind, J., McAllester, D.: Screamer: A portable efficient implementation of nondeterministic common lisp. Ircs technical reports series (1993)
37. Steele, G.: Common LISP: the language, 2nd edn. Digital Press (1990)
38. Sztipanovits, J., Karsai, G.: Model-integrated computing. Computer **30**(4), 110–111 (1997)
39. Wainer, G., Al-Zoubi, K., Dalle, O., Hill, D., Mittal, S., Risco-Martin, J., Sarjoughian, H., Touraille, L., Traore, M., Zeigler, B.: Discrete Event Modeling and Simulation: Theory and Applications, chap. DEVS Standardization: Ideas, Trends and Future (2010)
40. White, S., Sleeman, D.: Constraint handling in common lisp. Department of Computing Science Technical Report AUCS/TR9805, University of Aberdeen (1998)
41. Wilson, M.: Six views of embodied cognition. Psychonomic Bulletin & Review **9**(4), 625–636 (2002)
42. Zeigler, B., Chi, S.: Model-based architecture concepts for autonomous systems design and simulation. In: An introduction to intelligent and autonomous control, pp. 57–78. Kluwer Academic Publishers (1993)
43. Zeigler, B., Hammonds, P.: Modeling & simulation-based data engineering: introducing pragmatics into ontologies for net-centric information exchange. Academic Press (2007)
44. Zeigler, B., Luh, C., Kim, T.: Model base management for multifacetted systems. ACM Transactions on Modeling and Computer Simulation (TOMACS) **1**(3), 195–218 (1991)
45. Zeigler, B., Mittal, S., Hu, X.: Towards a formal standard for interoperability in m&s/system of systems integration. In: GMU-AFCEA Symposium on Critical Issues in C4I (2008)
46. Zeigler, B., Praehofer, H., Kim, T.: Theory of modeling and simulation: Integrating discrete event and continuous complex dynamic systems, 2nd edition edn. Academic Press (2000)