

implement its own behavior models. There is typically no reuse of models or code. This easily leads to high costs and potential inconsistent behaviors between simulators. This situation reminds us of the early days of simulation, where every platform made its own 3D models, built its own terrains and created its own scenario files. Exchange of models, terrains or platforms between simulations was not possible. Today there are a number of standards, such as COLLADA (Arnaud and Barnes, 2006), OpenFlight (OpenFlight, 2008) and MSDL (MSDL, 2008) that are used more and more to exchange this type of data between simulations. For example, in the case of terrain data, solutions exist to automatically generate terrain datasets for simulators based on a single common geo-referenced source, as depicted in Figure 1, thereby drastically reducing the effort needed (Kuijper et.al., 2007) and leading to highly correlated terrains.

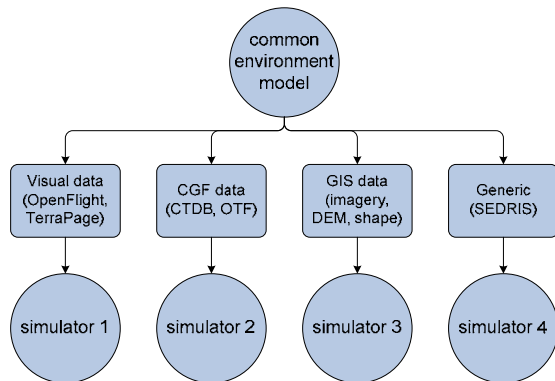


Figure 1: Generating terrain data from a common source.

For behavior specifications however, no standardized formats exist and reusing specifications or generating behavior specifications for different simulators from a common description seems a far cry from today’s practice.

1.3 Levels of reuse

This leads to the central question addressed in this paper: “can we make progress in reusing behavior specifications”? In order to answer this question, we will first define different levels of reusability, as illustrated in Figure 2.

Level 0: the behavior implementations are totally embedded in the simulation engine (i.e. hardcoded) and cannot be changed.

Level 1: as level 0, but the simulation environment provides an editor in which some parameters of the behaviors can be altered, e.g. the amount of damage a unit will accept before fleeing.

Level 2: as level 1, but the behavior specifications are defined using a scripting language and can be edited to change the entity behavior. The scripting language is

specific for the simulation engine. And the used AI paradigm, e.g. a Finite State Machine (FSM) or the Belief-Desire-Intention (BDI) model, is fixed.

Level 3: as level 2, but the behavior specifications are defined using an engine-independent language, e.g. Lua or Python. These behavior specifications have the potential to be shared with other simulators with the same AI paradigm.

Level 4: as level 3, but in this case not only the behavior specification are externalized but also the underlying AI paradigm. These behavior specifications are only domain dependent and therefore they can be reused relatively easy with simulators in the same domain.

Externalizing the AI paradigm means that the tactical, higher-level reasoning is performed by a specialized software component. This component uses (tactical) behavior specifications and performs decision making. It delegates lower-level actions, e.g. *move*, to the simulation engine. A similar architecture, in which Goal-Oriented Action Planning (GOAP) AI was interfaced with Unreal Tournament 2004, was developed by Pittman (2008).

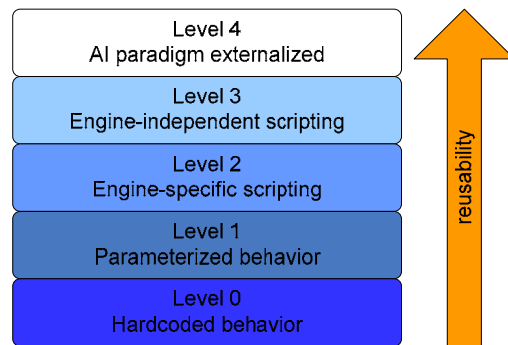


Figure 2: Reusability levels.

This paper describes our investigation into the practical possibilities regarding these reuse levels. Our research is based on experiments that involved the implementation of CISs in a number of different simulation environments, and we present our conclusions regarding the re-use level which seems most practical for our current purposes.

The remainder of this paper is organized as follows. Section 2 describes our architecture and approach for finding a solution to the “behavior reuse problem”. Next, Section 3 describes our implementation experiments and results. Section 4 presents our conclusions and way forward.

2. Behavior modeling approach

This section describes the main parts of our behavior modeling approach, which comprise a global vision and architecture as well as derived implementation-oriented goals and requirements.

2.1 Vision and architecture

Our aim is to develop a mechanism for behavior specifications for CGFs that are well-structured, composable, re-usable and repeatable. To achieve such behavior specifications for tactical doctrine we believe it is required to de-couple the behavior specification from the behavior execution engine. Given the four different levels of such de-coupling as described in the previous section, we ultimately aim at achieving level 4. This achieves great re-usability by explicitly modeling many behavior-related aspects including the AI paradigm itself. However, given the capabilities of current state-of-the-art simulation environments, our current work is an example of level 3 which is characterized by leveraging so-called engine-independent scripting. We de-couple behavior specifications from the execution engine, or in other words, we take the behavior specifications out of the execution environment and develop and maintain them externally. The behavior specification scripts are engine-independent, which means that they are not defined in a simulator-specific language. Instead, they are defined in a general purpose language, which enables that the behavior specifications may be *re-used* across different simulation environments. Figure 3 depicts a global architecture that embodies our more practical vision.

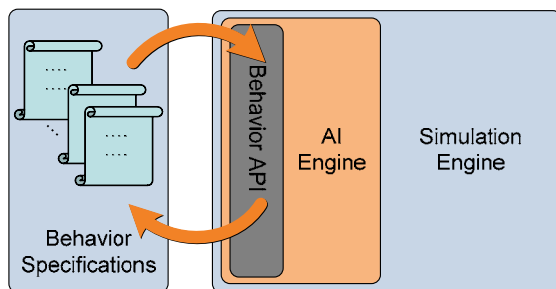


Figure 3: Behavior specification architecture.

The behavior specifications are in this case defined externally, i.e. outside the simulation engine. They interface to the simulation engine through a Behavior API, and they are executed by the AI engine (which normally interacts with other components of the simulation engine).

The Behavior API provides the basic functionality that is needed for controlling the entity that is modeled. For example, for an entity Soldier, the Behavior API may include the function *move(a, b)* to instruct the Soldier to move from *a* to *b*, and *find_cover(..)* to instruct the Soldier to find cover (based on certain parameters).

The behavior specifications define the intended entity behavior using

- the basic functionality of the Behavior API,
- the programming constructs of the used scripting language.

Typical mechanisms of a scripting language include an event mechanism, which allows the entity to respond to

events that occur, and other programming patterns that provide control of the entity.

Together the Behavior API and the scripting language provide the components that allow the *composition* of behavior specifications.

The use of scripting languages is a concept that is currently well-known and commonly applied in the game developer's community. For example, the scripting language Lua (Lua, 2008) is a programming language, designed as a scripting language that is used in various open source and commercial games. Lua is a lightweight, imperative language with extensible semantics. In particular the feature of extensible semantics makes it highly suitable for such applications.

We anticipate that our approach has the following main advantages and disadvantages.

Advantages

- Separating the behavior specifications from the simulation engine enables that behavior specifications can be adapted quickly, e.g. without time consuming compilation of source code of the simulator.
- Scripts could potentially be re-used between simulations.
- Combining a Behavior API and scripting language allows the definition of behavior specifications that are well-structured, composable, re-usable and repeatable.
- Scripting languages, like Lua, are compact and powerful, which implies that they are relatively easy to understand and to use. This could include Behavior Subject Matter Experts without any specific computer software background.

Disadvantages

- Using scripting languages can cause computational overhead. For example, in order to reason about its options an entity requires information about its environment, or more precisely, it needs to perform observations. Such observations (e.g. obtaining all visible friendly, neutral or hostile entities) need to be requested through the Behavior API regularly, which causes computational overhead.
- Scripting languages are not conducive to best practices in software engineering and code structuring, and they are generally harder to debug.

2.2 Goals and Requirements

To deliver on our vision, we have worked out the main goals and requirements of the components in the architecture depicted in Figure 3.

The simulation engine (or the AI engine within it) must implement the functions of the Behavior API adequately. In our CIS case for example, the simulation engine must implement the *move* function for an entity Soldier such that the Soldier moves realistically from *a* to *b*. Notice that what is considered ‘realistic’ or not, is likely to differ from application to application. A realistic implementation of a *move* function implies that environmental conditions, such as terrain and threat, possibly influence the execution of the *move* function. Also the implementation must include a backtracking mechanism in case destination *b* cannot be reached.

In practice, instead of developing a new simulation engine, an engine is usually selected from a number of existing simulation engines. It is critical to select a simulation engine that meets the relevant behavior requirements. The Behavior API must support the problem at hand. On the one side, it must provide sufficient control over an entity. On the other side, however, it should not provide too low-level functionality or parameters. In our case, where we are studying CISs, a *move* function is very common, whereas functions for *path planning* and *steering behavior* are considered to be too detailed. Hence the Behavior API should provide sufficient entity control without exposing functions that are too detailed. Note that when APIs are not exactly as needed, a middleware layer could provide extra functionality or conversions in order to make the API fit.

As mentioned before, the envisioned architecture allows that the Behavior API functions can be used in different specifications to compose structured behavior specifications that are re-usable. Additionally, it should be possible that these behavior specifications are in turn re-usable in other simulators. This can for example be achieved by using the Lua programming language. This approach allows for defining a hierarchy of behavior specifications that demonstrate coherent behavior.

3. Implementation Experiments

In order to support the conceptual approach presented in Section 2, proof-of-concept implementations of a scenario composed of a selection of CISs were made, both on commercial and in-house developed simulation engines. For this purpose we used the commercially available CryENGINE2 and Virtual Battle Space 2 (VBS2), and the in-house developed TNO Enhanced Virtual Environment (EVE). The objective of our three implementations is to find out how well modern simulation engines are able to deal with behavior defined at the level of a CIS. By comparing the results of the implementations we are able to render a judgment on how reusable the behavior specifications are across different engines.

3.1 Scenario Setup

The execution of CIS behavior poses constraints and requirements on the API of the simulation engine. What these constraints and requirements are follows directly from the abstraction level at which a CIS is defined: A CIS is a formal specification of a combat doctrine written in natural language. There are three mayor aspects that characterize a CIS: 1) A CIS describes the behavior of a group of entities, not a single entity, 2) a CIS is able to activate other CISs if required to achieve its goal, and 3) a CIS can be interrupted by so called *situational interrupts*, usually as a result of events that occur in the environment.

As described earlier in Section 2.2 the Behavior API of the simulation engine should support the problem at hand. Hence, the Behavior API should enable the requirements of the behavior defined at the level of a CIS, being: 1) performing actions, 2) obtaining information from the environment, and 3) notifying when events occur in which the CIS is interested. The implemented scenario should cover at least all of these requirements.

The scenario that we used is composed of two CISs: An infantry squad of four members, divided over two teams, performs the CIS *Move bounding overwatch* while advancing to a certain position. While doing so, they are subject to a mortar attack, and therefore execute the CIS *React to indirect fire*. When the indirect fire has stopped, the squad re-evaluates or continues the bounding overwatch movement.

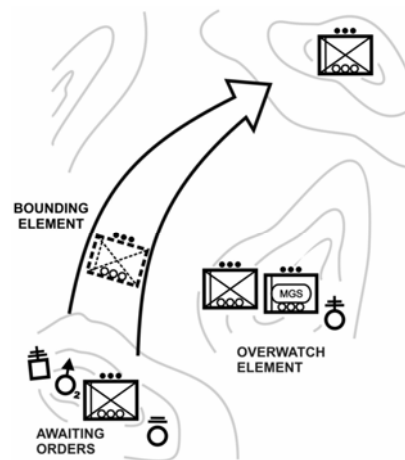


Figure 4: The *Move bounding overwatch* maneuver. The bounding element advances to a concealed position while the overwatch element provides cover from a concealed position. Taken from FM3-21.11: The SBCT Infantry Rifle Company (2003).

The CIS *Move bounding overwatch* describes how an infantry squad traverses open terrain when hostile contact is expected and speed is not essential, see also Figure 4. *Move bounding overwatch* is characterized

by the bounding movement of one team (bounding element), while the other team provides cover from a concealed position (overwatch element). This CIS requires both performing actions that are coordinated between the teams and using information from the environment, e.g. to find a concealed position to give cover from.

The CIS *React to indirect fire* is an example of a CIS executed as a reaction to a situational interrupt, in this case when subject to artillery or mortar attack: all squad elements immediately take cover within a certain maximum radius. When the indirect fire has stopped the squad will switch back, if applicable, to doing what it was doing before the indirect fire attack.

Although the two CISs *Move bounding overwatch* and *React to indirect fire* do not cover all possible interaction between the behavior specification and the simulation engine, all requirements of the Behavior API for the execution of CISs are apparent.

3.2 Virtual Battlespace 2 (VBS2)

The first simulation engine in which we implemented the scenario is Virtual Battlespace 2 (VBS2). VBS2 (Bohemia Interactive, 2008) is a commercial serious game based on the first person shooter Armed Assault. VBS2 is, as stated earlier, already being used by the RNLA for training purposes.

VBS2 provides the ability to design scenarios using a visual scenario editor in combination with its own scripting language. Since VBS2 is aimed at the military domain the collection of functions that the scripting language provides can be considered very complete: it contains functionality not commonly found in games, e.g. for hierarchical group coordination.

To implement the scenario we used VBS2's engine-specific script language and the built-in Finite State Machine (FSM) functionality. The FSM is a commonly accepted AI paradigm that defines behavior in relation to the *mental state* (Tozour, 2004), or similarly to a *context* (Gonzalez, 2008), of an entity. We were able to control an infantry squad and have it execute an FSM that implements the bounding overwatch movement. We also created an FSM that finds cover for each unit.

Although we succeeded in implementing the scenario there were several difficulties. First, simulating the indirect-fire attack in the scenario is cumbersome: Since VBS2 does not support the creation of user-defined events we had to develop a mechanism that allows us to notify the FSM that an event had occurred. This issue applies to all events that are not covered by the limited set of built-in events. Second, VBS2 provides no control over the FSM itself after it is

started: One cannot determine whether the FSM is still running, or in which state it currently is. As a result, for example, automatically switching back to the previous behavior after the indirect fire has stopped becomes practically impossible. Summarizing, VBS2 provides an API that is quite complete. However, the FSM functionality does not provide sufficient control for behavior modeling.

3.3 CryENGINE2 (Crysis)

The CryENGINE2 (Crytek GmbH, 2008) is an example of a modern simulation engine. The AAA game *Crysis* was built on this engine. The CryENGINE2 is supplied with a level editor, in which, besides levels, CGF behavior can be graphically designed using flow graphs. In addition to flow graphs, CGF behavior can also be implemented using Lua script files. We used Lua for our prototype.

The structural breakdown of a behavior for an entity in the CryENGINE2 is depicted in Figure 5: On the highest level the behavior is called a *Character*, which in its turn is composed of one or more *Behaviors* (analogous to the *state* of a FSM). At any given time only a single behavior of a character is active and receives events, which can be built-in or user defined. As a result of an event the character can switch its behavior, or activate any of the built-in or custom designed *goalpipes*. A goalpipe is a (recursive) collection of atomic actions, like *move* or *fire*.

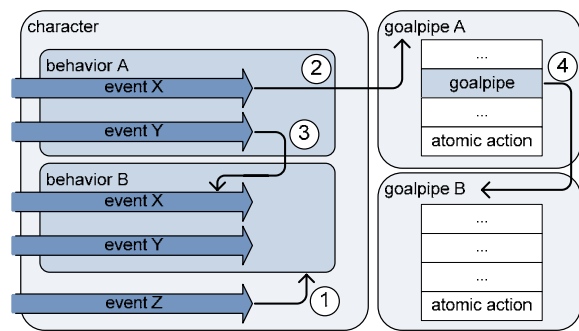


Figure 5: The structural breakdown of behavior in CryENGINE2. (1) Events trigger a change of behavior, (2) as a result of an event a goalpipe is activated, (3) another event is triggered, (4) besides atomic actions a goalpipe can also contain other goalpipes.

The scenario was implemented using the characters and behavior approach described above. We created a character where every CIS was defined as a behavior. Coordinated by user-defined events an infantry squad uses a bounding overwatch drill to advance to a certain position. Other events (built-in) were used to simulate indirect fire, which triggered the characters to switch behavior and activate a goalpipe in order to find cover.

In order to switch back to the previous behavior when the indirect fired has stopped, the CryENGINE2 provides a specific language element.

The chosen approach proves successful. The combination of characters, behaviors, goalpipes, and flexible events works well. A drawback is that a behavior implementation in CryENGINE2 is for a single entity, instead of for an entire group: This decentralized approach requires (user-defined) events for group coordination. Given a CIS, a centralized approach would be preferred.

3.4 TNO Simulation Framework EVE

The third and last simulation engine in which the scenario described in Section 3.1 was implemented is TNO's Enhanced Virtual Environment (EVE). EVE is a modular, data-driven simulation framework intended for fast prototyping of new applications. Among other features, EVE provides 3D visualization as well as many subsystems for low-level steering and physics. The behavior subsystem for EVE was designed from scratch, so special attention could be given to the behavior modeling paradigm and the choice of scripting language.

For the modeling of the behavior we chose to use a combination of hierarchical and stacked (Fu, 2004), (Tozour, 2004) FSM approaches. The reasons for this choice are twofold: First, the behavior described in a CIS is applicable to a group of entities, instead of a single entity. By using a hierarchical approach a group of entities (e.g. a platoon) can be instructed to execute a CIS by calling a single FSM. The FSM, in its turn, is responsible for assigning specific behaviors to (groups of) entities, which can be executed by sub-FSMs. The second reason considers the *situational interrupts* that cause other CISs to become active for a certain period of time. By using a stacked approach a new CIS can temporarily become active, while the previous CIS is pushed on top of the stack. When the new CIS ends and is no longer active, the old CIS will be popped from the stack and continue its execution.

The scripting language that we choose to use is again Lua. Besides the advantages of Lua mentioned in Section 2.1 Lua was easy to integrate with EVE, which acts as its host application.

Using the approach described above the scenario was implemented for EVE. State machine scripts were designed for both CISs *Move bounding overwatch* and *React to indirect fire*, as well as a default behavior, as shown in Figure 6. In these scripts events are triggered (*situational interrupts*), and the entire group executing the script will respond. For example, when an *indirect-fire* event is received the group will activate the react-to-indirect-fire script. In this script all group members

will query the simulation engine (EVE) for their nearest cover location, and move there.

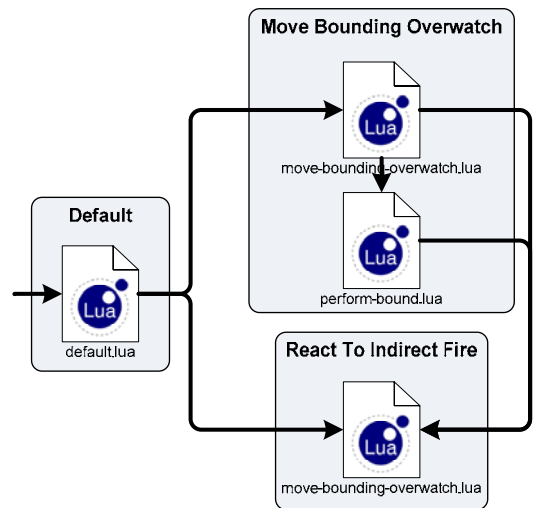


Figure 6: The behaviors implemented in the EVE behavior subsystem, and how they relate. Note that the *Move Bounding Overwatch* FSM contains a sub-FSM that is executed by a team to perform a single bound movement.

The approach chosen for the modeling of the scenario in EVE proved successful. By using hierarchical and stacked FSMs, in combination with events, the CISs can be implemented while maintaining the original structure within and between CISs.

3.5 Discussion

When comparing our three implementations (illustrated in Figure 7) we find similarities, but also differences that concern the reusability of the behavior implementations. VBS2 is categorized as level 2, since it uses a engine-specific scripting language. The CryENGINE2 and our EVE implementation can both be categorized as level 3: they use an externalized engine-independent scripting language, but the AI paradigm is not externalized. Although the scripting language and even the AI paradigm of both are the same, the behavior implementations are hardly interchangeable. This is due to the differences in the details of the AI paradigm and Behavior API, as explained next.

All simulation engines we used rely on some form of the finite state machine AI paradigm. However, there are differences in the way they are applied: First, in the CryENGINE2 each entity has its own behavior, whereas in EVE and VBS2 a behavior controls an entire group of entities. Second, our EVE implementation and the CryENGINE2 use events to control the execution of behavior specifications, whereas VBS2 uses a polling mechanism. And third, the engines differ in the flexibility they provide. For



Figure 7: Illustrations of our implementations in VBS2 (left), CryENGINE2 (middle) and EVE (right)

example in VBS2 one can also create a behavior for a single entity or even abandon the use of the FSM. On the other hand, in the CryENGINE2 one is forced to use the character-behavior-event-goalpipe functionality.

The functionalities that the engines provide in the Behavior API differ as well. VBS2's API is complete for the military domain: it contains numerous functions for controlling e.g. formations, command hierarchy, and radio communication. However, it sometimes lacks control. The CryENGINE2's API is less complete (more generic), but provides more control, although at the price of complexity. An example of this is the function to find cover: VBS2's *findCover* finds cover within a specified range, with optional line of sight to a target. CryENGINE2's *hide* action can find cover within a specified range using more than 30 methods (*nearest*, *nearest-backwards*, *nearest-to-target*, *behind-vehicles*, *most-front-left-of-target*, et cetera).

In order to achieve a higher level of reusability it is advisable to strive for more standardization of both the AI paradigm and the Behavior API. A major step towards this objective is to take the AI paradigm out of the simulation engine (reusability level 4), which reduces the dependencies between the behavior specification and the simulation engine. The behaviors themselves can then be specified in a reusable format, utilizing home-made programs or AI middleware like Soar, ACT-R, AI.implant and Kynapse, which interface with the standardized Behavior API. A promising approach to developing a standardized Behavior API is to base it on existing standards. For certain applications, the Battle Management Language (BML) (SISO C-BML, 2008) draft standard is a good candidate in the tactical domain. BML promises an unambiguous and standardized language for communicating plans and orders to live, simulated, and robotic forces.

Until the moment arrives that most simulation engines have adopted a common standardized Behavior API, an intermediate solution could be considered. This solution would use a standardized behavior

specification as input for a process of code generation for those engines that do not support the standardized Behavior API. However, this approach can become a daunting task.

4. Conclusions and Way Forward

This paper has investigated the possibilities for developing and re-using behavior specifications. It focused on representing so-called Combat Instruction Sets (CISs) which are formal behavior specifications in natural language. We investigated the implementation of such specifications in different gaming and simulation environments. For this purpose we used the commercially available Virtual Battle Space 2 (VBS2) and CryENGINE2, and the in-house developed TNO Enhanced Virtual Environment (EVE). We studied in particular the possibilities and obstacles for re-use of behavior specifications in these environments. We experienced that the environments strongly differ in the possibilities they provide, and the limitations they impose for developing behavior specifications.

Based on our experiences we have established that the following factors are key-elements in establishing re-use of behavior specifications.

- *The degree of de-coupling of the behavior specifications from the simulation engine*
A strong de-coupling from the engine that is used is required, and preferably a well-known and accepted specification language (scripting).
- *The availability of a well-defined Behavior API*
A (standardized) Behavior API is required that functionally matches the problem domain at hand and that is implemented by different simulation engines.
- *The level of control that is provided by the applied AI paradigm*
The applied AI paradigm (e.g. stacked finite state machines) provides a certain level of control that can be employed in the behavior specifications. For many serious applications (e.g. implementing tactical doctrine) the required level of control is high.

The factors mentioned above point in the direction of open architectures and systems. Such an approach opens up possibilities that many current simulation engines do not yet provide for developing and re-using behavior specifications. For example, the AI paradigm itself may be taken out of the simulation engine and be implemented externally using a scripting language (reusability level 4). A prerequisite for applying this approach successfully is that a standardized Behavior API is available. We suggest basing such a Behavior API on e.g. the Battle Management Language (BML) for the tactical domain. As an intermediate solution, before such a standardized Behavior API is available, a code generation process could be used to make a common behavior specification fit on specific simulation engines. This approach may however prove very challenging. Nonetheless, we certainly believe that a more open approach is the way forward for achieving higher levels of re-use.

5. References

- Arnaud, R., Barnes M.C. (2006). *COLLADA: Sailing the Gulf of 3d Digital Content Creation*. Wellesley MA: AK Peters, Ltd.
- Bohemia Interactive (2008), Retrieved from the web November 20, 2008, <http://www.bistudio.com/simulations.html>
- Crytek GmbH (2008), Retrieved from the web November 20, 2008, <http://www.crytek.com>
- eSim Games (2008), Retrieved from the web November 20, 2008, http://www.esimgames.com/steel_beasts_pro.htm
- FM3-21.11: The SBCT Infantry Rifle Company (2003), Retrieved from the web November 20, 2008, <http://www.globalsecurity.org/military/library/policy/army/fm/3-21-91/c03.htm>
- Fu, D., Houlette, R. (2004), The Ultimate Guide to FSMs in Games. In S. Rabin (Eds), *AI Game Programming Wisdom 2*, (pp 283-302). Hingham, Massachusetts: Charles River Media, Inc.
- Gonzalez, A., Stensrud, B., Barrett, G. (2008). Formalizing context-based reasoning: A modeling paradigm for representing tactical human behavior. *International Journal of Intelligent Systems*, vol 23 (7), 822-847.
- Kuijper, F., Van Son, R., Van den Heuvel, F. (2007). Rapid Modelling of Urban Mission Areas Using Ground-Based Imagery, Proceedings of the 2007 Fall Simulation Interoperability Workshop, Orlando, US, 07F-SIW-029
- Lua (2008). Retrieved from the web November 20, 2008, <http://www.lua.org/>
- MSDL (2008). Retrieved from the web November 20, 2008, <http://www.sisostds.org/index.php?tg=fileman&i>

- [dx=get&id=5&gr=Y&path=SISO+Products%2FSISO+Standards&file=SISO-STD-007-2008.doc](http://www.sisostds.org/index.php?tg=fileman&id=5&gr=Y&path=SISO+Products%2FSISO+Standards&file=SISO-STD-007-2008.doc)
- OpenFlight (2008). Retrieved from the web November 20, 2008, <http://www.multigen-paradigm.com/products/standards/openflight/>
- Pittman, D. (2008) Command Hierarchies Using Goal-Oriented Action Planning. In S. Rabin (Eds), *AI Game Programming Wisdom 4*, (pp 383-392). Hingham, Massachusetts: Charles River Media, Inc.
- SISO C-BML (2008), Retrieved from the web November 20, 2008, http://www.sisostds.org/index.php?tg=fileman&id=33&gr=Y&path=&file=CBML_Spec_Final_1_29_08_v0.09.doc
- Tozour, P. (2004). Stack-Based Finite-State Machines. In S. Rabin (Eds), *AI Game Programming Wisdom 2*, (pp 303-306). Hingham, Massachusetts: Charles River Media, Inc.

Author Biographies

DR. KLAAS JAN DE KRAKER is a member of the scientific staff at TNO Defence, Security and Safety. He holds a Ph.D. in Computer Science from Delft University of Technology. He has a background in Computer-Aided Design and Manufacturing, collaboration applications, software engineering (methodologies), meta-modeling and data modeling. Currently he is leading various simulation projects in the areas of simulation based performance assessment, collective mission simulation, multifunctional simulation and serious gaming.

PHILIP KERBUSCH, MSc. is a member of the scientific staff at TNO Defence, Security and Safety. Philip holds a MSc. degree cum laude in Artificial Intelligence from Maastricht University. He has a background in computer science and knowledge engineering. His current work involves various applications of artificial intelligence and 3D visualization in the defense and safety domain.

ERIK BORGERS is a senior scientist in the M&S department at TNO Defence, Security and Safety in the Netherlands. He has a professional background in Electronic Engineering and Artificial Intelligence (AI). Erik leads the tactical modelling sub-section, which focuses on realistic and reusable entity models, including the use of AI. Erik has been active as a project lead and architect during the production of constructive and virtual staff trainers both for the RNLA and for training civil Disaster Management Teams. His current work focuses on the use of Agent technology for kinetic and non-kinetic simulations, interoperability and Serious Games.