Automated Techniques for Generating Behavioural Models for Constructive Combat Simulations*

Matt Selway, Kerryn R. Owen, Richard M. Dexter, Georg Grossmann, Wolfgang Mayer and Markus Stumptner

Abstract Constructive combat simulation is widely used across Defence Science and Technology (DST) Group, which typically execute behavioural models written in a scripting or programming language. This representation is authored and maintained by software developers for a specific combat simulation and is time-consuming, requires specific expertise, and can lead to inconsistencies between the same behaviour in different simulations. Furthermore, it is desirable to engage military subject matter experts in the elicitation and verification of behaviours and this requires a representation that is both comprehensible to a non-programmer and translatable to different simulation execution formats. This paper presents the Hierarchical Behaviour Model and Notation (HBMN) behaviour representation that fulfils these requirements, along with processes for translating written military doctrine texts to HBMN and from

Matt Selway
University of South Australia, e-mail: Matt.Selway@unisa.edu.au

Kerryn R. Owen
Land Simulation, Experimentation, and Wargaming, Land Capability Analysis, Joint and Operations Analysis Division, Defence Science and Technology Group, e-mail: Kerryn.Owen3@dsto.defence.gov.au

Richard M. Dexter
Land Simulation, Experimentation, and Wargaming, Land Capability Analysis, Joint and Operations Analysis Division, Defence Science and Technology Group, e-mail: Richard.Dexter@dsto.defence.gov.au

Georg Grossmann
University of South Australia, e-mail: Georg.Grossmann@unisa.edu.au

Wolfgang Mayer
University of South Australia, e-mail: Wolfgang.Mayer@unisa.edu.au

Markus Stumptner
University of South Australia, e-mail: Markus.Stumptner@unisa.edu.au

*Preprint. Paper accepted and presented at the 24th National Conference of the Australian Society for Operations Research. Publication in a volume of Springer's LNMIE series to be confirmed.

HBMN to executable simulation behaviours.
HBMN incorporates aspects of existing business process and behaviour representations and provides a hierarchical schema that, for specifying behaviours at varying levels of abstraction, allows an incremental approach to developing and refining behaviour models from partial models to concrete executable models. Natural Language Processing and Information Extraction techniques are used to automatically generate initial HBMN models from military doctrine texts. Following this, a model transformation framework allows the specification and execution of transformations from HBMN into other models, in particular simulation execution formats. The transformations are specified as visual contracts utilising the notation of the underlying source/target models. This provides a cohesive solution to modelling combat behaviours across all stages of development.

1 Introduction

Constructive combat simulation is widely used across DST Group in support of operations research studies. Such simulations typically execute behavioural models written in a scripting or programming language that is authored and maintained by software developers dedicated to a specific combat simulation. This approach is time consuming, requires specific expertise, and can create inconsistencies in representations of the same behaviour between multiple simulations. Furthermore, it is a desirable practice to engage military subject matter experts in the elicitation and verification of behaviours representing military doctrine and tactics (see [4, 5, 12]). This means there is a requirement to have a representation of behavioural models that is not specific to a particular simulation, is comprehensible to a non-programmer but translatable to a simulation execution format, and offers an intuitive interface as presented in [6].

The University of South Australia and DST Group have developed the Hierarchical Behaviour Model and Notation (HBMN) behaviour representation, an automated process for translating written military doctrine texts to HBMN, and an automated process for generating executable behaviours from HBMN to other modelling formats. This workflow is illustrated in Fig. 1.

The HBMN representation adheres to a strictly hierarchical schema allowing behaviours to be expressed at varying levels of abstraction. Therefore, HBMN models can be developed incrementally in a top-down approach by refining abstract partial behavioural models into concrete behaviours before execution in an existing simulation environment. The HBMN representation incorporates selected elements of existing business process and behaviour representations, including Business Process Execution Language [13], Business process Model and Notation [14], and Hierarchical Task Networks [7, 8].

HBMN Models can be automatically generated from military doctrine texts via the following process. Natural Language Doctrine Analysis (NLDA) is performed using Natural Language Processing and Information Extraction techniques [15] using rule-

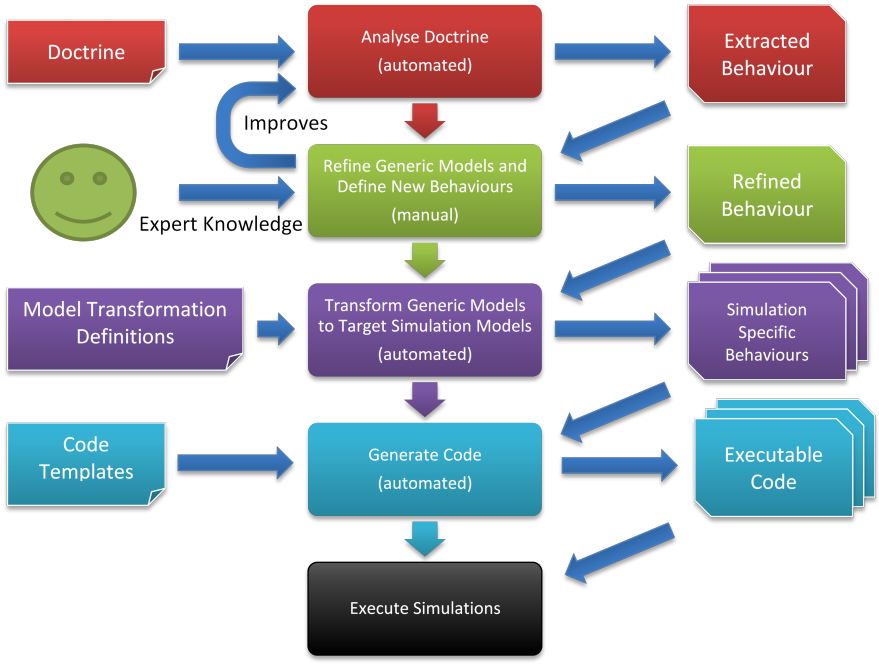


Fig. 1 Behaviour Extraction and Modelling Workflow

based methods and general purpose lexical resources (i.e. dictionaries incorporating additional information), such as [10], to perform the extraction. A key feature is the contextual integration of analysed sentences. By integrating sentences through context, more complete models can be produced instead of producing disconnected fragments.

HBMN Models can be automatically transformed into other simulation formats using the Model Transformation and Code Generation framework (MTCG). This framework takes an input model, executes a defined set of transformation rules, and outputs the results in a target model. The framework is based on the use of visual contracts for defining the preconditions, post-conditions, and invariants that should hold for the execution of the model transformation [9]. The transfer and manipulation of this data between source and target models is defined using Visual Transformation Operators: an extensible set of operators for data manipulation built on a core library of operators [1].

This paper presents the HBMN schema and compares it some of the more common behaviour representation schemas. In addition, the Natural Language Doctrine Analysis and the Model Transformation processes are introduced.

2 Hierarchical Behaviour Model and Notation

A behaviour representation appropriate for the specification of behavioural models for combat simulations in a simulation agnostic way needs to fulfil a number of requirements. These include the need to represent typical behaviour control-flow (such as sequential, parallel and exclusive branch activities), resource sharing, and information transfer, as well as support for modularity to allow the reuse of models. These requirements have evolved from those as described in [3]. Existing behaviour representations do not adequately cover all of the requirements, hence, the Hierarchical Behaviour Model and Notation (HBMN) was developed. HBMN incorporates selected elements of existing business process and behaviour representations, primarily:

- BPEL (Business Process Execution Language) [13]
- BPMN2 (Business process Model and Notation version 2) [14]
- HTN (Hierarchical Task Networks) [7]

Behaviours represented in HBMN adhere to a strictly hierarchical schema so that they can be expressed at varying levels of abstraction. The development of HBMN models can be performed incrementally using a top-down approach where abstract partial behavioural models are refined into concrete executable behaviours that can be run in existing simulation environments. HBMN models can also be developed in a bottom-up approach where a behaviour model is developed by combining previously defined lower-level models in new contexts.

Concrete executable behavioural models are developed through the collaboration of subject matter experts, such as military commanders who specify behavioural models, and technical experts who refine abstract behaviours into detailed executable simulations. HBMN facilitates this collaboration by providing a visual representation for modelling suited to non-programmers, and an encoding in a machine-readable format that enables automated processing of the models by software tools.

A hierarchical component-oriented approach to behaviour modelling enables modular composition of behaviours from existing behaviour modules which may be shared in a library of common behaviours and reused in subsequent models. Each behaviour module is described by a well-defined interface capturing essential properties such as entry and exit conditions, data input and outputs, and resource requirements. A common interface enables the composition of modules and facilitates the analysis and transformation of behavioural models to other representations: for example, different visual and textual views of a model or program code for specific target simulation environments.

This approach has the potential to promote consistency of behavioural models across simulations as well as streamline the implementation of simulations for a variety of target simulation environments when combined with model transformation capabilities. Benefits of the strictly structured model are the following:

- The model eases interpretation and translation to and from other representations, including text and code.

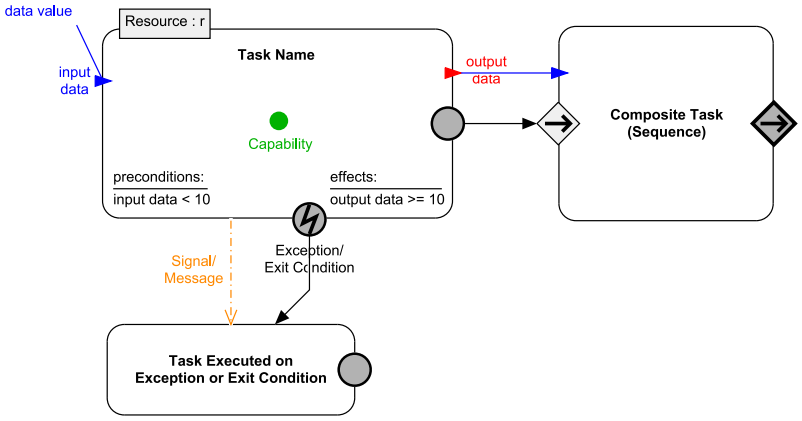


Fig. 2 Common HBMN Task Elements

- Compositionality and comprehensive interfaces facilitates reuse of behaviour components.

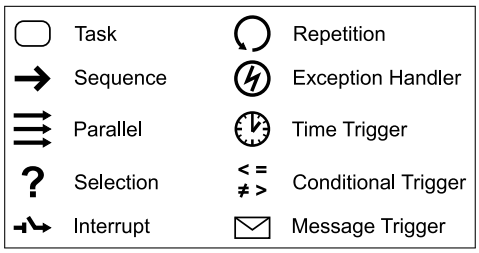
The behaviour model is uniform across all modelling levels, from abstract to fully executable. It combines reactive modelling (that is, looking only at the current state as opposed to planning ahead), hierarchical task network methods (hierarchical decomposition, preconditions and effects), and prioritised and interruptible tasks. The following section outlines the basic elements of the HBMN schema.

2.1 HBMN Schema

HBMN models are composed of tasks using behaviour *combinators* (a term adopted from Process Algebras such as CSP [11]) to form composite behaviours. Tasks are either primitive tasks that have corresponding implementations in a simulation environment (possibly derivable through a transformation), or abstract tasks which may be realised through one or more composite behaviour models. Each task, whether it is primitive or composite, has a common interface. The main features include a label, one or more exit conditions, data inputs/outputs, and required resources and their capabilities. In addition, each task can be associated with preconditions that control the activation of the task, effects that enforce post-conditions on the execution, data values for data inputs that are known at design time, and signals or messages which support communication. The common elements are shown in Fig. 2.

Primitive tasks are illustrated as a rounded rectangle with a circular exit condition (no icon), while composite tasks use diamond gateways containing an icon indicating the type of task (e.g., a sequence uses a single arrow, a parallel has a triple arrow, and selection has a question mark, see Fig. 3).

Fig. 3 Task Type Icons



Each task has a distinct exit condition that denotes the successful completion of the task, which may or may not be named. Exit conditions are the dark grey gateway for composite tasks. Additional (named) exit conditions represent alternative outcomes and exceptions that may have occurred during the execution of the task. All exit conditions may be the source of flow arcs denoting the subsequent task to be executed.

Input and output data parameters (blue and red triangles, respectively) can be associated with a task and are described by a label (unique among all data elements associated with the task) and a type (drawn from a taxonomy of data types). Moreover, data values can be explicitly supplied for input parameters if they are known in the context of the behaviour being specified. Data flows (blue lines) are used to connect explicit values to inputs, as well as connecting inputs and outputs between tasks. In addition, data flows may connect two inputs (or two outputs) together when modelling nested behaviours: the inputs of the higher-level task are mapped to inputs of the nested task(s) and vice versa for outputs. Data types and data flows (from values or outputs) are required for all data parameters of tasks in executable models.

Resources required by a task can be specified through resource parameters which define a label and the required resource type in the form '`Type : label`'. Resource parameters may denote individual entities (e.g., a soldier), or aggregate entities such as squads. Moreover, the capabilities requested from the resource can be specified in terms of a shared capability model. Capabilities provide a locking mechanism on resources to prevent the possibility of tasks being executed concurrently by a single resource; for example, if they require the same capability.

A task may be associated with precondition expressions (logical expressions formed from the input parameters of the task) that govern if a task may execute. While multiple precondition expressions can be specified, they are treated as a logical conjunction. Therefore, all the precondition expressions taken together must be satisfied before a task can be executed. While executable models require fully specified logical expressions, more abstract models can be provided with a description of the precondition. This facilitates understanding by subject matter experts, and refinement of the simple description into logical expressions can later be performed by technical experts.

Similarly, effects can be specified that govern the expected results of the task execution, i.e. logical expressions formed from the output parameters of the task.

The logical conjunction of the effect expressions is expected to be satisfied as a result of the successful completion of the task.

Tasks can send and receive signals to other tasks. These can consist of arbitrary messages to other resources or tasks and typically block the execution of a task until the required signal is received.

2.1.1 Execution Semantics

A task can only be executed if the following conditions are met:

- its control entry point is enabled,
- all data input parameters have received a value,
- all required resource parameters have been assigned a resource,
- the capabilities of all required resources are available,
- any signals that are present have been received, and
- all specified precondition expressions are satisfied.

Resources (and their capabilities) are acquired when a task commences execution and released when the task ceases execution. Data output parameters are considered local to the task until the task completes. Values are propagated to the enclosing scope once a task has (successfully) completed execution. Signals are emitted on successful completion of a task.

If an exception or alternative exit condition is encountered the output data values are propagated to the exception handler if they are of interest to it (by being assigned the appropriate input data parameter). Otherwise, they are assigned a special nil value, and signals are not emitted. The exception handler can then choose to propagate the data value further and/or emit signals as required.

2.1.2 Combinators

In total there are 9 combinators, or composite task, types. For brevity we will highlight five major combinators: Sequence, Parallel, Repetition, Selection, and Interruption. For each type of combinator data inputs and outputs, exit conditions, and resource parameters are formed as the union of the respective parameters and conditions of the sub-tasks.

The most basic composite task is a Sequence, in which a number of sub-tasks are ordered by sequence flow connections between the exit-point of one task and the entry point of the next. A sequence completes once all of its subtasks have completed successfully (or an error handler leads to successful completion). In contrast, the Parallel combinator allows the representation of tasks that are performed concurrently with synchronisation: either the same resource performing concurrent actions or multiple resources (e.g. each soldier in a squad) performing actions at the same time. The parallel task completes once all of the sub-tasks have completed.

If an unhandled exit condition causes early termination of the parallel tasks, the incomplete subordinate tasks are terminated as well.

Task repetition (or loops) can be represented using the Repetition combinator in which a task is executed repeatedly in sequence governed by a loop condition. Loop conditions can control the repetition by specifying a logical expression (which will cause the repetition to occur for as long as it evaluates to true or until it evaluates to true), a specific number of repetitions (possibly the result of a calculation), or based on the result of executing a separate subordinate task. To support different types of repetitive behaviour, the output parameters of a repetition can output values after each repetition or they can be combined using an aggregation expression and output once when the repetition task completes. For example, this allows a repetitive task to continuously output values to another task running in parallel.

The Selection combinator allows a subordinate behaviour to be chosen from a set of alternatives. Unlike choices in other languages such as BPMN [14], the selection combinator does not perform an exclusive choice by default. Instead, the preconditions, etc. are checked and all enabled subordinate tasks are executed. In contrast to a parallel combinator, subordinate tasks cannot be enabled and executed while some tasks are still executing. Moreover, once the executing tasks are completed the selection task completes. However, a termination condition can be provided that specifies the number of subordinate tasks to be executed. If the number of tasks required has not been met after some tasks have completed, the preconditions of remaining tasks are rechecked and any newly enabled tasks are executed until the required completion condition has been met. In addition, the subtasks of a selection combinator can be given a priority, which is used to determine which task(s) will be executed when more tasks are enabled than can be executed (otherwise the choice of task(s) to execute is random).

Finally, HBMN provides an explicit representation of interruption behaviour through the Interruption Combinator. Interruptions are similar to Selections with priority, with a few key differences: only one subtask executes at a time; a task with higher priority than the currently executing task can become enabled and, if so, the current task is suspended and the higher priority task is executed; and once a subtask completes, the next highest priority suspended task is resumed. This is highly useful for modelling combat behaviour (e.g., where a squad behaviour must patrol an area, react to contact with an enemy, and then return to patrolling once the contact has been resolved) as it allows the direct representation of suspending and resuming some task without relying on complex combinations of loops and selections. As a result it reduces the complexity of the models and improves their clarity for subject matter experts.

2.2 Comparison with other Behaviour Representations

In general, HBMN provides several advantages over other approaches to modelling behaviour. These include:

- A strictly hierarchical, well-structured model with comprehensive interface definitions covering control, data, and resource aspects of primitive and composite behaviours
- Support for reactive behaviours including interruptions and prioritisation
- A layered approach that supports incremental collaborative definition of behaviours
- Simpler visual notation than comparable process modelling approaches (BPMN2, BPEL, YAWL)

The following briefly discusses the limitations of particular process/behaviour modelling approaches.

BPEL [13] is a language designed for orchestrating Web Service processes. It combines graph-based and structured hierarchical modelling primitives in a single language. The resulting language possesses complex execution semantics where control flow, exception handling, and data flow interact. Behaviour prioritisation, interruptions, resource sharing, and reactive features are not easily modelled in the language. The language is formally defined in terms of XML Schema, and mappings from visual notations to BPEL exist. Although interface specifications provide uniform descriptions of individual service operations, the underlying Web Service XML stack is exposed to the user and the web service centric model does not easily translate to hierarchical behaviour definitions.

BPMN2 [14] is a rich language for modelling conceptual business processes and workflows. However, the number of behaviour primitives is large and aspects of the language lack a universally agreed upon execution semantics. The graph-based modelling approach requires careful thought to avoid unstructured behaviour models that are difficult to analyse and reuse. Simple hierarchical refinement of individual tasks is possible, yet resource sharing and the absence of a comprehensive data flow model make modular composition difficult. The complexity of the language renders any form of analysis and translation to other representations challenging. Reactive behaviour traits are restricted to simple event based triggers; interruptions are not covered.

Hierarchical Task Networks (HTN) are one of the main approaches in automated planning [7, 8]. HTN planners refine abstract task graphs into more detailed (executable) plans using a library of available methods that may realise each abstract task. This approach is advantageous as domain-specific sub processes can be incorporated easily. HTNs can represent concurrent behaviours but are not well suited for reactive behaviours where behaviours may be interrupted and prioritised dynamically.

(E)FFBDs and SysML/UML Activity Diagrams are two languages for modelling engineering functional decomposition of systems and processes [2]. Both languages can capture control flow, concurrency, data flow, (informal) triggering conditions, and multi-instance tasks. The languages are expressive yet care must be taken to avoid unstructured models which would be difficult to analyse and translate. Advanced reactive concepts, such as interruptions and signals are not directly expressible in the language and must instead be simulated through other mechanisms.

3 Natural Language Doctrine Analysis (NLDA)

To support the development of behavioural models in a timely fashion, we have developed automated processes to extract HBMN models from military doctrine texts. This NLDA process uses Natural Language Processing and Information Extraction techniques to identify tasks, resources, data parameters, and data values in doctrine texts. To perform the process extraction, we use rule-based methods based on a generalisation of previous work into the extraction of structural models from software requirements documents [15]. The extraction framework makes use of general purpose lexical resources, such as WordNet and VerbNet, through a common interface (Uby [10]) to support the extraction of behaviours from scratch. In addition, it integrates knowledge gained from previously extracted and refined models to improve the extraction of new models over time. That is, it reuses existing HBMN models and their components to enhance newly extracted models. The goal is to reduce the amount of manual refinement required as a repository of reusable high-quality models is developed.

4 Model Transformation and Code Generation (MTCG)

We have incorporated a Model Transformation and Code Generation (MTCG) framework to automate the process of generating simulation specific models and code from the HBMN models. The MTCG framework takes an input model, executes a defined set of transformation rules, and outputs the results in a target model—this could be the same or a different type of model to that which was input. The framework is based on the use of visual contracts for defining the preconditions, post-conditions, and invariants that should hold for the execution of the model transformation [9]. The contracts define patterns of elements that are to be matched in the source and/or target models, as well as constraint expressions that the matched elements must satisfy for the contract to be fulfilled; an example is shown in Fig. 4. When used for the *execution* of transformations, the target model patterns are instantiated with information carried across from model elements that match the source patterns. The transfer and manipulation of this data between source and target models is defined through the use of Visual Transformation Operators [1]: an extensible set of operators for data manipulation built on a core library of operators. Moreover, complex transformation definitions can be constructed hierarchically, allowing defined transformations to be modularised and reused. Figure 5 illustrates a contract with a nested transformation definition that copies the value of the source parameter to the target.

The use of visual contracts and operators for model transformations enables the transformations to be defined using the native notations of the source and target models. This allows transformation specification to be separated from transformation implementation; simpler and more concise definitions of transformations; and a major component of the transformation, the source and target patterns, to be developed by those familiar with the underlying notations. In addition, the checking

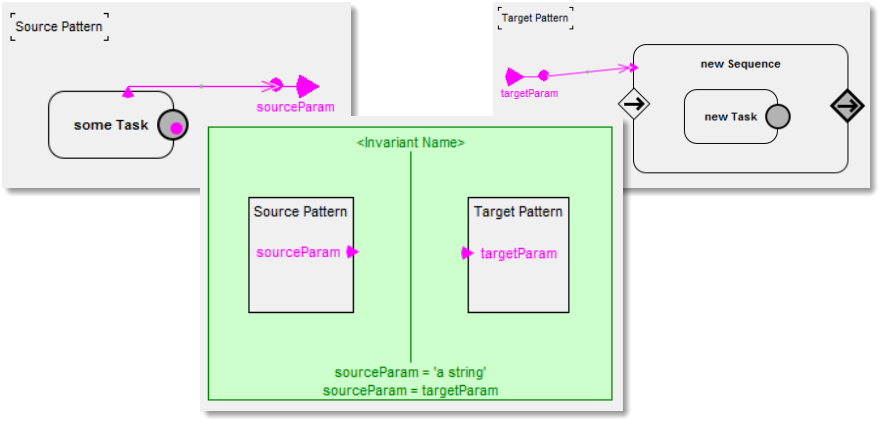


Fig. 4 MTCG Positive Invariant (centre) with the Source Pattern definition (left) and Target Pattern definition (right). The invariant defines a simple equality constraint between the parameters of the source and target patterns. The source and target pattern definitions utilise the concrete model notation augmented with mapping parameters (pink nodes).

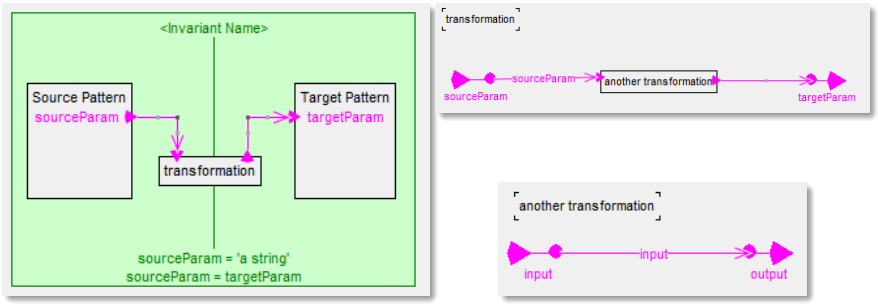


Fig. 5 MTCG Positive Invariant with Transformation Definition (left) and the hierarchical decomposition of the transformation (top-right and bottom-right).

of models can be performed independently from the transformation execution; for example, to check that the models conform to the pre-/post-conditions or that a pair of models satisfies the transformation conditions. This is useful to check previously transformed models against a modified transformation definition. Furthermore, the structure of the transformation rules allows the transformation itself to be checked for correctness, with various inconsistencies and their causes able to be reported to the transformation designer. These features facilitate the long term maintenance and update of transformations by different people at different times and by those more familiar with the source and target notations than specific transformation notation.

5 Conclusion and Future Work

In modelling behaviours for constructive combat simulations, there is a need for a simulation agnostic modelling language that can be transformed into executable formats, facilitates collaboration with subject matter experts, and directly represent reactive aspects such as interruptions. Such a representation would reduce the time taken to develop simulations and minimise inconsistencies between different simulations. The Hierarchical Behaviour Model and Notation presented here, and the accompanying Natural Language Doctrine Analysis and Model Transformation and Code Generation processes, address these concerns. It provides a simplified and well-structured behaviour representation that covers control, data, and resource aspects of primitive and composite behaviours, as well as direct representations of reactive behaviours such as interruptions. Furthermore, it supports the incremental definition of behaviours from abstract, including initial models automatically extracted from doctrine texts, to concrete definitions that can be transformed to executable formats using the Model Transformation and Code Generation framework.

The HBMN Schema, NLDA and MTCG processes are packaged in the HBMN Modeller software and further refinement of the software and techniques used is ongoing. At this stage, basic doctrine has been translated using NLDA and the MTCG is capable of producing executable code for the COMBATXXI simulation. Future work includes ongoing refinement of the NLDA and MTCG, building a library of HBMN behaviours, and disseminating the model to a wider user community.

References

1. Berger, S., Grossmann, G., Stumptner, M., Schrefl, M.: Metamodel-based information integration at industrial scale. In: Proc. ACM/IEEE International Conference on Model Drivel Engineering Languages and Systems (MODELS), pp. 153–167 (2010)
2. Bock, C.: UML 2 activity model support for systems engineering functional flow diagrams. Systems Engineering **6**(4), 249–265 (2003). DOI 10.1002/sys.10053
3. Bowden, F.D.J., Davies, M.: Application of a role-based methodology to represent command and control processes using extended petri nets. In: Proc. 1997 IEEE International Conference on Systems, Man and Cybernetics. Orlando, Florida (1997)
4. Bowden, F.D.J., Davies, M., Dunn, J.M.: Representing role based agents using coloured petri nets. In: 5th Computer Generated Forces and Behavioral Representation Conference (1995)
5. Cox, A., Gibb, A., Page, I.: Army training and CGDs – a UK perspective. In: Proc. 5th Computer Generated Forces and Behavioral Representation Conference (1995)
6. Dexter, R.M., Piotto, J., Pash, K.M.: Bridging the gap: A generic behaviour framework for behaviour representation (extended abstract). In: Proc. MODSIM 2013. Australia (2013)
7. Georgievski, I., Aiello, M.: HTN planning: Overview, comparison, and beyond. Artificial Intelligence **222**, 124–156 (2015). DOI 10.1016/j.artint.2015.02.002
8. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
9. Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W.: Automated verification of model transformations based on visual contracts. Automated Software Engineering **20**(1), 5–46 (2013). DOI 10.1007/s10515-012-0102-y

10. Gurevych, I., Eckle-Kohler, J., , Hartmann, S., Matuschek, M., Meyer, C., Wirth, C.: Uby: A large-scale unified lexical-semantic resource based on LMF. In: Proc. 13th Conference of the European Chapter of the Association for Computational Linguistics, pp. 580–590 (2012)
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)
12. Lewis, J.W.: Agents that explain their own actions. In: Proc. 5th Computer Generated Forces and Behavioral Representation Conference. Orlando, Florida (1994)
13. OASIS: Web services business process execution language version 2.0. Standard (2007). URL `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
14. OMG: Business process model and notation, version 2.0. Standard formal/2011-01-03, Object Management Group (2011). URL `http://www.omg.org/spec/BPMN/2.0`
15. Selway, M., Grossmann, G., Mayer, W., Stumptner, M.: Formalising natural language specifications using a cognitive linguistic/configuration based approach. Information Systems **54**, 191–208 (2015). DOI 10.1016/j.is.2015.04.003