Validating Planning Domain Models Using B-AMN

M. M. West, D. E. Kitchin and T. L. McCluskey
The School of Computing and Engineering,
The University of Huddersfield,
Huddersfield, UK

Abstract

The validation of planning domain models is an important issue and can present problems. In this paper we describe ongoing work which attempts to overcome these problems through the construction of a B-AMN specification which models the domain. The B-Method utilises B-AMN, a state-based formal specification language with tool support provided by the B-Toolkit. We describe how this tool support provides facilities for both animation and proof, and we propose the use of the B-Method at an initial level of domain capture. The approach is illustrated using a simple transport domain, where a high-level model, which can be reasoned with, is produced. In addition, animation allows validation by users.

1 Introduction

Much AI planning research has focused on improving the efficiency of planning algorithms, but the problems associated with modelling, encoding and validating an AI planning domain model also need to be addressed. Most would agree with Penix et al's remarks:

> A system with a totally correct reasoning algorithm will be ineffective if its model of the world is flawed. Therefore, validation becomes a critical task of evaluating a part of the system to be deployed. [12].

When AI planning researchers encode a new planning domain model their usual approach is to encode it immediately in either a domain definition language such as PDDL [1] or a domain modelling language such as OCL_h [6]. This model would then be validated by testing it with one or more planning engines on a number of tasks. Some researchers would also make use of tools, such as syntax and operator consistency checkers [7]. However, it is likely that any reasonably sized realistic domain model will continue to contain errors and inconsistencies for some time. A planner may manage to produce a solution despite the fact that the domain model is flawed. One approach to this is to use model checking for validation, as in [12], but this is limited by potential state space explosion, as is planning. Another approach could be to assume that the domain model will be incomplete as in the *SiN* algorithm [10]. This is a fruitful assumption in many ways, as philosophically no model can ever be 'proved' correct. However, this approach neglects the issue of correctness - the incomplete parts must still be validated and bugs identified and eliminated. In this paper we explore an alternative method of validation, using B-AMN (B Abstract Machine Notation) and the B-Toolkit [5]. The method is illustrated by a simple model of a transport planning domain, based on the *Towns* machine in [15]. We show how the approach we describe enables the development of a precise, high-level model which can be validated both by animation and proof. The advantage of animation within the B-Toolkit is that the user can validate the behaviour of the specification without having to access or understand the formal specification language.

2 Formal Methods in Software Engineering and Planning

Formal Methods in software engineering covers the capture and analysis of a (formal) specification of software within a structured formal language, as well as the refinement of a formal specification into an efficient implementation. The formal specification language has to be appropriate to the application at hand, be sufficiently abstract to allow formal reasoning, and be well supported with a tools environment. The primary concerns in formal methods are to show that the initial specification is internally consistent and externally valid, and to prove that that the derived implementation is correct with respect to the specification. These processes are meant to improve the quality of the software process and product, as they are aimed at the early identification and removal of bugs. There are a variety of specification languages, an important distinction being whether the language is explicitly based on algebra or whether it is based on an explicit notion of an abstract state [19]. Superficially at least, there is a strong similarity between *state-based* formal specification languages and planning languages. Both kinds of languages are designed to allow engineers to represent actions precisely and declaratively. For example, B-AMN and a STRIPS-language are both based around the notion of a state, allow the developer to create operators, and in both cases those operators are defined using pre- and postconditions. Further, they are both based on the assumptions of closed world, default persistence and instantaneous operator execution. The B-Method requires the creation of state invariants for validity and documentation purposes. State invariants are also used in some planning languages, though the function of invariants for validation and verification purposes has been extended to operational concerns such as plan generation speed-up [13].

The difference between those using formal specification to describe systems and those using a planning language to model a planning domain, is that in the former case the specification is used as a blueprint for design, whereas in the latter case the specification is used as input to a planner to be reasoned with in order to construct plans to achieve goals. States in languages such as B-AMN are built up from mathematical data types such as sets, relations, sequences etc. Thus B-AMN is richer expressively than a typical language used for encoding a planning domain model. With some exceptions, such as in deductive planning [3] , much of the work carried out in planning research assumes little or no structure to types.

3 Modelling a Transport Planning Domain

3.1 The B-Method

The B-Method was developed during the late 1980s by Jean-Raymond Abrial. The B-Method in common with other 'state based' methods utilises the concept of a *state machine* and B-AMN is a state-based formal specification language. AMN allows for static type checking of specifications, dynamic validation and mathematical verification by proof to ensure the correctness of the design process. Although B-AMN is itself non-executable, it is possible to ‘execute’ specifications in B-AMN to make their behaviours visible to the customer or user. (This type of specification execution is often termed *animation*.) The B-Method and notation can be used within the B-Toolkit. This provides support in the form of an interface, editors, syntax and type checkers, an animator, proof obligation generators and provers, and document production.

A specification will be constructed from one or more abstract machines, with the components of a machine being its variables, invariant, initialisation and operations. A typical abstract machine state comprises several variables which are constrained by the machine invariant and initialised. Operations on the state contain explicit preconditions; the postconditions are expressed as ‘generalised substitutions’, giving the language a ‘program-like feel’. For example, the postcondition of an operation which incremented a state variable $v1$, would be $v1 := v1 + 1$. Proof obligations check that, for example, the machine invariant remains true throughout the machine's operations. To illustrate the method we use a simple example, consisting of a single abstract machine: a fragment of a transport domain consisting of a number of places which may be linked by *roads*, and a number of objects which have to be transported between places via existing roads. Some objects are regarded as *mobile* and *person* is a class of mobile object. An example of a mobile non-person is *car* and an example of a non-mobile object is *tent*. This domain is similar to the planning Logistics domain included with the SHOP planner [11].

3.2 Specification of *TownsXXX*

The specification of the transport domain and possible states it may inhabit is accomplished in B-AMN by a state machine called *TownsXXX* where the variables and constants of *TownsXXX* are formed from *deferred sets* (i.e. not instantiated).[1] The sets of *TownsXXX* are $OBJECT$, $TOWN$ and constants and variables are derived from these sets via *types*. For example, *person* and *mobile* are both constants: $person \subseteq mobile$ where $mobile \subseteq OBJECT$. Variables form part of the *state* of *TownsXXX* and include

- *roads* and *Links* which are both relations between $TOWN$ and itself ($roads \in TOWN \leftrightarrow TOWN$ and similarly for *Links*). *Links* is a variable expressing all possible connections of roads between towns and is formed by the transitive closure of *roads*, where the idea for this was taken from [15];

- *at_town* is a function between an object and a town ($at\_town \in OBJECT \ \twoheadrightarrow\ TOWN$) which models an object's place, and captures the fact that an object can only be in one town at any one time;

- *Can_carry*, a relation between objects. $obj1 \mapsto obj2 \in Can\_carry$ means that $obj1$ can carry $obj2$. *Can_carry* is transitive, so if $Can\_carry(x) = y$ and $Can\_carry(y) = z$, then $Can\_carry(x) = z$;

- *Picked_up* is a function linking an object *obj1* to a set of other objects, meaning that *obj1* has picked up the set of objects. As objects are introduced to the functional domain of *Can_carry*, they are also introduced to the domain of *Picked_up* where they

[1] More information about *TownsXXX* is provided at http://helios.hud.ac.uk/scommmw/TownsXXX/ which contains postscript versions of machine and proofs, together with an animation script.

are initialised to the empty set. That is, all objects which can carry other objects are modelled so that they are immediately added to the picked_up function, but initially they will not have picked anything up. This ensures that only those objects able to carry another object can pick an object up.

The types of the variables of the machine are declared in an invariant which also contains other constraints on their values. For example the constraint on the values of functions *Can_carry* and *Picked_up* is expressed in the invariant of *TownsXXX* as:

$\mathsf{dom}\ (\ Can\_carry\ ) = \mathsf{dom}\ (\ Picked\_up\ ).$

The operations of *TownsXXX* potentially change its state and we view them as being of two kinds:

(i) Knowledge Accumulation

Each of the variables of the machine are initialised to the empty set and these operations provide a means of inputting knowledge into the planning domain. For example there is an operation which links two towns to form a road, and an operation for placing an unplaced object. This facility for knowledge accumulation (i.e. not relying on a single initialised domain) makes for greater generality. In planning, initial states would be defined as part of a planning problem to be solved. Here we can use knowledge accumulation operators to build up to *any* initial state. An example follows of an operation **parachute_in** which places an object at a town. In the example, *obj* , *town* are inputs and := means assignment, denoting the change in variable *at_town*.

```
 parachute_in ( $obj$ , $town$ ) =
PRE     $town \in TOWN \wedge obj \notin \mathsf{dom}\ (\ at\_town\ ) \wedge obj \in OBJECT$
THEN    $at\_town := at\_town \cup \{\ obj \mapsto town\ \}$
END
```

The preconditions for the above operation include the condition that knowledge is not overridden:

$obj \notin \mathsf{dom}\ (\ at\_town\ )$

means that before the operation *obj* is not attached to a place.

(ii) Performance of Tasks

These operations perform tasks, for example pick up an object or move an object between towns. The result of the operation **drop_object** is that *obj2* is dropped by *obj1* (which it has previously picked up).

```
drop_object ( $obj1$ , $obj2$ ) =
   PRE     $obj1 \in \mathsf{dom}\ (\ Picked\_up\ ) \wedge obj2 \in OBJECT\ \wedge$
           $obj2 \in Picked\_up\ (\ obj1\ )$
   THEN    $Picked\_up := Picked\_up \mathbin{<\!\!\!+} \{\ obj1 \mapsto Picked\_up\ (\ obj1\ ) - \{\ obj2\ \}\ \}$
   END
```

Picked_up links an object *obj1* to a set of other objects. In the above, *obj2* is dropped and is removed from the set. However no other variable is altered, so that *obj1* and *obj2* remain in the same place. There are no assumptions in the specification about any ordering of the operations. Thus knowledge can be accumulated between tasks. The preconditions for the tasks effectively orders them. Thus an object must have been picked up before it is dropped: $obj2 \in Picked\_up\ (\ obj1\ )$.

3.3 Validation of the Transport Domain

TownsXXX was validated using the complementary activities of animation and proof [9]. Formal proof compensates for the fact that tests used for animation can seldom be exhaustive. On the other hand there is no use in seeking a formal proof of a property of a specification if counter examples indicate that the property is not present. The proof engine of the B-Toolkit uses backwards and forwards inference, and rewriting is treated as a special form of backwards inference and is used for animation.

3.3.1 Animation

An advantage of the B-Toolkit is that it provides a tool which allows the developer to animate a specification. During animation, the deferred sets will be instantiated via the tool, and operations can be executed on the state. The interface shows the values of variables before and after execution, allowing validation by the user. In order to check that the *TownsXXX* machine behaved as expected it was animated, and deferred sets were instantiated so that *TOWN* was provided with set members *Huddersfield*, *Bradford*, *Leeds*, *etc* and OBJECT with members *alice*, *fred*, *car*, *bike*, *tent*, *rucksack*, *etc*. Subsets {*alice*, *fred*, *car*, *bike*}, {*alice*, *fred*} model *mobile*, *person* respectively. Knowledge was accumulated by, for example, linking towns to form roads and placing objects. The results of the animation can be output to a script (and one is available from the web page). Animation showed that the machine captured the limited transport domain. The following example of animation shows the *before* and *after* states of the *car* dropping the rucksack. In the before state, the *car* carries the *rucksack* and *fred*, and all are at *Bradford*. In the after state, the *car* is carrying only *fred*, but the places of *car*, *rucksack* and *fred* are unaltered – all remain at *Bradford*.

```
Current State /* before the operation */
/* Links is omitted as it is lengthy */

  roads     {Hudds |-> Bradford , Hudds |-> Leeds}
  at_town   {car |-> Bradford , rucksack |-> Bradford , elephant |-> Hudds ,
                     alice |-> Hudds , fred |-> Bradford}
  Can_carry {elephant |-> rucksack , fred |-> rucksack , car |-> fred ,
                     elephant |-> car , car |-> rucksack , elephant |-> fred}
  Picked_up {fred |-> {} , elephant |-> {} , car |-> {rucksack , fred}}
========================================================
drop_object ( obj1=car , obj2=rucksack )

Current State /* after the operation */

  roads     {Hudds |-> Bradford , Hudds |-> Leeds}
  at_town   {car |-> Bradford , rucksack |-> Bradford , elephant |-> Hudds ,
                        alice |-> Hudds , fred |-> Bradford}
   Can_carry {elephant |-> rucksack , fred |-> rucksack , car |-> fred ,
                        elephant |-> car , car |-> rucksack , elephant |-> fred}
 *Picked_up {car |-> {fred} , elephant |-> {} , fred |-> {}}
========================================================
```

This facility for animation is similar to a *stepper* in a planning tool support environment, such as the one in GIPO [16] and has much the same effect: it provides an excellent way of identifying and removing bugs.

3.3.2 Proof

The B-Toolkit has a facility for generating proof obligations (theorems) for checking consistency. For example a proof obligation is generated which checks that the invariant is obeyed *initially.* Also for every operation, *op*, assuming that the machine invariant, *inv*(*TownsXXX*), and operation precondition, *pre* (*op*), is true, the condition must remain true after each operation. For example a proof obligation associated with *drop_object* is that assuming that *inv* (*TownsXXX*) and *pre* (*drop_object*) are true, the domain of *Picked_up* is (still) the same as the domain of *Can_carry*.

$$\begin{array}{l} \mathit{inv}\ (\ \mathit{TownsXXX}\ ) \wedge \mathit{pre}\ (\ \mathit{drop\_object}\ ) \Rightarrow \\ \quad \mathsf{dom}\ (\ \mathit{Can\_carry}\ ) = \mathsf{dom}\ (\ \mathit{Picked\_up} \mathbin{<\!\!\!+} \{\ \mathit{obj1} \mapsto \mathit{Picked\_up}\ (\ \mathit{obj1}\ ) - \{\ \mathit{obj2}\ \}\ \}\ ) \end{array}$$

Proof obligations are discharged either automatically or with some interaction. In all 24 proof obligations were generated, of which 16 were discharged automatically, 4 interactively and 4 proof obligations (currently) remain undischarged.

4 Discussion

We would envisage using the B-Tool at an initial level of domain capture, to produce a high-level model. The production of such a high-level model helps clarify thinking about a domain and helps develop intuitions about it in much the same way as it can during software engineering requirements specification. We can also reason about this model at an early stage and at different levels. This can be done independently of any particular planning engine, whereas typical AI planning domain validation requires the use of a planner. The B-Tool aids user or domain expert validation of the model as the animator provides a simple interface for the domain expert to see the effect of applying operators directly on the state without going through a possibly lengthy plan generation process. It also means that the domain expert does not need to understand the details of the specification language itself. The B-Tool can produce both proof obligations and proofs which demonstrate the consistency and validity of the model. The animator provides validation that is complementary to proof, while the proofs demonstrate consistency (or the lack of it). Formal proof can be expensive and time consuming. Animation tests specific cases and can be made understandable to a customer or a user of the implemented system. This means that potential misunderstandings as to the functionality of the system can be avoided. However testing can seldom be exhaustive so that proof provides for the general case. Having validated the model by both animation and proof obligation generation and discharge, the specification would then provide the basis for the development of a more detailed implementation-level model.

The development using the B-Tool allows a high-level exploration of the workings and modelling of the domain. While this may not be vital in small domains, potentially it may be very valuable in large ones. As is often the case in the use of formal methods in software engineering, the approach we present here may be particularly useful in the modelling of planning domains for safety-critical applications, or for high integrity systems, such as space applications. The B-Method and AMN have been successfully used in the specification of a number of systems, including the SACEM system to control train speeds on the RER Line A in Paris [4] and an automatic train control system [17]. The B-Method was also used for the design and validation of the transaction mechanism for smart card applications [14]. As security in smart card applications is paramount, the use of the B formal method gives confidence and provides mathematical proof that the design of the transaction mechanism fulfills the security requirements. B-AMN was also used to specify a computer system which supported the collection and organisation of data for the French population census of 1991 [2]. This data was used for statistical purposes and was critical for the success of the census process. The use of formal methods in knowledge engineering has been advocated by [20], and [8] also argue that the use of formal specifications for the verification and validation of knowledge-based systems will improve the quality of those systems. The perceived advantages of using tool-supported formal methods are:

- Lack of Ambiguity - Models expressed in a formal specification language have a precise syntax and semantics based on mathematics, which eradicates the ambiguity found in natural language

- Design - We see the B AMN specification as providing a high level model which can be used to capture unambiguously the details of a new domain. After validation this can be used as the basis for an implementation in a language that can be input to a planner

- Validation and Verification - In this case via the animator and proofs

- Evaluation and Maintenance The existence of a precise, high-level specification would allow new domain features to be added or altered in the future, and the model re-tested for consistency and validity, as it is a simple matter to regenerate proof obligations.

5 Conclusions and Future Work

This paper work describes work in progress. Future work might be as follows:

- Investigation of the process of implementation. By this we mean the translation from the AMN specification into a language suitable for input to a planner. The B-Toolkit supports the processes of refinement and implementation of an abstract machine, but the target language would be a programming language such as C++ or Java. We need to test whether the translation from AMN to PDDL or OCL_h is straightforward, and if it preserves the qualities of the formal specification.

- Specification and implementation of a larger example. This would allow us to test the whole process from specification to implementation. Validation of the specification via a non-planning expert and via proofs would be further investigated for effectiveness.
- Some work has been carried out into the automatic synthesis of domain-dependent planners from specifications [18]. It would be interesting to investigate the possibility of a similar synthesis of domain model encodings from formal specifications.

In conclusion, the development of a high-level model in B-AMN using the B-Toolkit may be particularly useful in the development of domain models for large, complex and/or safety-critical applications. It allows the modeller to develop a precise, unambiguous specification, which can be validated by the user via the animator. Further validation can be performed via the generation and discharge of proof obligations. The tool support provided by the B-Toolkit means that the user does not need to be familiar with B-AMN. User validation of a planning domain model is not easy without some form of tool support, as it is unlikely that domain experts would be familiar with or fully understand, for example, pieces of ADL or an HTN operator. One of the reasons often cited for formal methods not being employed in the development of software engineering systems is that the formalisms are not understandable to most stake-holders in the proposed system. The approach we describe here offers the advantages associated with the use of formal methods, plus easier validation by domain experts who are not required also to be experts in either formal methods or AI planning.

References

[1] AIPS-98 Planning Competition Committee. PDDL - The Planning Domain Definition Language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[2] P. Bernard and G. Laffitte. The French Population Census for 1991. In J. P. Bowen and M. G. Hinchey, editors, *ZUM'95 – 9th International Conference of Z User's, September 1995, Limerick, Ireland*, pages 334 – 352. LNCS 967, Springer-Verlag, 1995.

[3] S. Biundo and W. Stephan. Modeling Planning Domains Systematically. In *Proceedings of the 12th European Conference on Artificial Intelligence*, 1996.

[4] G. Guiho and C. Hennebert. Sacem software validation. In *Proceedings of 12th International Conference on Software Engineering*, pages 186–191. IEEE Computer Society Press, 1990.

[5] B-Core (UK) Ltd. http://www.b-core.com/.

[6] T. L. McCluskey. Object Transition Sequences: A New Form of Abstraction for HTN Planners. In *The Fifth International Conference on Artificial Intelligence Planning Systems*, 2000.

[7] T. L. McCluskey and D. E. Kitchin. A Tool-Supported Approach to Engineering HTN Planning Models. In *Proceedings of 10th IEEE International Conference on Tools with Artificial Intelligence*, 1998.

[8] P. Meseguer and A. D. Preece. Assessing the role of formal specifications in verification and validation of knowledge-based systems. In S. Bologna and G. Bucci, editor, *Procedings of the Third International Conference on Achieving Quality in Software*, pages 317–328. Chapman & Hall,London, 1996.

[9] P. Mukherjee. Computer-aided validation of formal specifications. *Software Engineering Journal*, 10(4):133–140, July 1995.

[10] H. M. Munoz-Avila, D. W. Aha, D. Nau, R. Weber, L. Breslow, and F. Yaman. SiN: Integrating case-based reasoning with task decomposition. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 999–1004, 2001.

[11] D. Nau, Y. Cao, A. Lotem, and H. Munoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, 1999.

[12] J. Penix, C. Pecheur, and K. Havelund. Using Model Checking to Validate AI Planner Domain Models. In *Proceedings of the 23rd Annual Software Engineering Workshop, NASA Goddard*, 1998.

[13] J. M. Porteous. *Compilation-Based Performance Improvement for Generative Planners.* PhD thesis, Department of Computer Science, The City University, 1993.

[14] D. Sabatier and P. Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science (Springer-Verlag)*, pages 348 –368. Springer Verlag, 1999.

[15] S. Schneider. *The B-Method: An Introduction.* Palgrave, 2001.

[16] R. M. Simpson, T. L. McCluskey, W. Zhao, R. S. Aylett, and C. Doniat. GIPO: An Integrated Graphical Tool to support Knowledge Engineering in AI Planning. In *Proceedings of the 6th European Conference on Planning*, 2001.

[17] I. H. Sorensen and D. Neilson. The specification of an automatic train control system. B-Core(UK) Ltd, 2002.

[18] B. Srivastava and S. Kambhampati. A Structured Approach for Synthesizing Planners from specifications. In *Proceedings of the 12th IEEE International Conference on Automated Software Engineering*, 1997.

[19] J. G. Turner and T. L. McCluskey. *The Construction of Formal Specifications: an Introduction to the Model-Based and Algebraic Approaches.* McGraw-Hill Software Engineering Series, London. ISBN 0-07-707735-0, 1994.

[20] F. van Harmelen and D. Fensel. Formal Methods in Knowledge Engineering. Technical report, The University of Amsterdam, 1995.