Identifying Domain Invariants from an Object-Relationship Model

Tim Grant

Netherlands Defence Academy
Faculty of Military Sciences, P.O. Box 90.002, NL-4800PA Breda, The Netherlands
tj.grant@nlda.nl

Abstract

Many planning algorithms require domain constraints, axioms, or invariants as a part of the domain model. This paper presents the Domain Invariant Extraction Technique (DIET) for identifying binary invariants systematically from an object-relationship domain model. The technique can be driven by an oracle (the real-world domain or a domain expert) or traces (from instructors, planners, or control system execution). It has been implemented as part of a larger system for inducing planning operators, and used for eight domains. The paper describes the object-relationship-invariant ontology and the technique itself, giving examples of its use and identifying potential applications. Evidence is given to show that DIET should be extendible to the object-attribute-value ontology. Further work to be done includes extending DIET to higher-arity invariants.

Background

Many planning systems make use of domain constraints, axioms, invariants, or exclusion relations, e.g. Stefik (1981), Fox (1983), Chapman (1987), Drummond (1987), Dean, Firby & Miller (1988), Drummond & Currie (1989), Tenenburg (1991), Bresina, Drummond & Kedar (1993), Grant (1995), Blum & Furst (1997), McCluskey & Porteous (1997), Zhang & Foo (1997), Gerevini & Schubert (1998), Kautz & Selman (1998), Fox & Long (1998), Rintanen (2000), Lin (2004), Mukherji & Schubert (2005), and Kuter, Levine, Green, Rebguns, Spears & DeJong (2007). Georgeff (1987, p.367) emphasized the role of constraints when he defined plan generation – which he terms plan synthesis – as follows:

> Plan synthesis concerns the construction of some plan of action for one or more agents to achieve some specified goal or goals, given the constraints of the world in which those agents are operating.

In the literature a variety of terms is used for domain constraints. Drummond & Currie (1989) called them "invariants". Bresina et al (1993) apparently used "domain constraints" and "behavioural constraints" interchangeably. Tenenberg (1991) employed "static axioms" to ensure that only valid state descriptions (which he called "legal situations") were used in plan generation.

We shall use *constraints* as the generic term, and *invariants* to mean constraints that must remain true throughout the execution of a plan from initial to goal state for the plan to be valid[1]. Constraints may apply to states (e.g. McCluskey & Porteous, 1997; Fox & Long, 1998; Rintanen, 2000), to actions (e.g. Stefik, 1981; Chapman, 1987; Dean et al, 1988), or – as in GraphPlan (Blum & Furst, 1997) - to both states and actions. In the database literature (e.g. Nijssen & Halpin, 1989), constraints are also known as *validation rules* and invariants are known as *static constraints*. Not all constraints can be depicted in graphical conceptual schemata, but may have to be represented as logical formulae, tables, or graphs.

Constraint satisfaction approaches predominate in the scheduling literature. In planning, constraints may be used to speed up plan generation, to extract optimal parallel plans, or to convert a planning problem into a satisfiability equivalent (e.g. Refanides & Sekallarion, 2009). Constraints may be provided by knowledge engineers or domain experts. Several researchers have developed algorithms for discovering or synthesizing constraints. Most synthesize constraints from planning operators (e.g. Rintanen, 2000), some from state descriptions (e.g. Lin, 2004; Mukherji & Schubert, 2005), and others (e.g. Gerevini & Schubert, 1998) from both the operator-set and the initial state.

By contrast, in our research we are interested in extracting domain invariants from a design specification of the domain expressed in terms of the objects or entities to be found in the domain together with the relationships between them. In many applications (e.g. those that are safety critical), it is important that the extraction process is systematic, i.e. the complete set of invariants is extracted

[1] Some authors (e.g. Rintanen, 2000) define invariants as facts that hold in all states that are reachable from an initial state by the application of a number of operators. This assumes that the initial state is valid. For example, if an object is defined as being in two locations simultaneously in the initial state, then subsequent actions and states may well be invalid, resulting in an *impossible world* (Nijssen & Halpin, 1989).

from the available information. We draw on insights from the database and object-oriented design literature.

A justification for our approach is that domain modellers may not have access to information about the domain's behaviour, i.e. information on domain states and actions. Firstly, the domain may not yet exist in the real world. For example, designers of an oil refinery will want to be sure before construction starts that it will operate effectively without violating safety constraints. Secondly, it may be difficult to obtain perfect information about an existing, real-world domain. The difficulty may be deliberate or a matter of practicality. For example, the designers of a new luxury car will want to deny information about its capabilities reaching unauthorised recipients, such as competitors or journalists. They may want to deliberately disguise their new product so that competitors will gain a false (or at least incomplete) impression of its capabilities. Practical difficulties in obtaining perfect information about a real-world domain can be time, space, resource, cultural, or other limitations and barriers.

This paper presents the Domain Invariant Extraction Technique (DIET) for identifying invariants systematically from an object-relationship domain model. DIET can be driven by an oracle, such as a domain expert or observations of a real-world domain, or by extracting the needed information from traces obtained from an instructor, a planner, or control system execution. Our contribution is two-fold:

(1) We add a fourth usage of constraints: to aid in the specification or synthesis of a domain model.

(2) DIET derives invariants from the domain objects and their inter-relationships.

The following sections summarize the object-relationship-invariant ontology and its basis in the literature, describe how a domain is represented using this ontology, detail DIET and illustrate its use, outline possible applications, draw conclusions, and outline further work to be done. The blocks world is used for illustration.

Ontology

The representation used in DIET is based on the entity-relationship-constraint ontology from the database literature, extended using object-oriented concepts. Domains are represented using an extension of Chen's (1976) Entity-Relationship Model (ERM) in which Chen's entity-sets and entities are regarded as object-classes and -instances, respectively. In the blocks world, for example, the domain model might identify `Hand`, `Block`, and `Table` object-classes, with `block1`, `block2`, and `block3` as instances of the `Block` class.

In ERM, relationships are directed, with each relationship linking a pair of objects (-classes/-instances). Hence, relationships are binary. In object-oriented terms, they represent *instance relationships* that can be true at one point in time and false at another. For example, if `hand1` is holding `block2` at a particular moment, then the `holding` relationship linking the `hand1` object to the `block2` object is true. When `hand1` puts `block2` down or stacks it on another block, then the `holding` relationship between them becomes false. If `hand1` then picks up `block3`, then the `holding` relationship between `hand1` and `block3` becomes true. Figure 1 depicts the Nilssen (1980) blocks world in the ERM graphical notation.

In Chen's (1976) ERM notation, relationships can have cardinality constraints, so that a relationship can be one-to-one, one-to-many, many-to-one, or many-to-many. In the traditional blocks world (e.g. as described by Nilsson, 1980), a hand can only hold one block at a time, and a

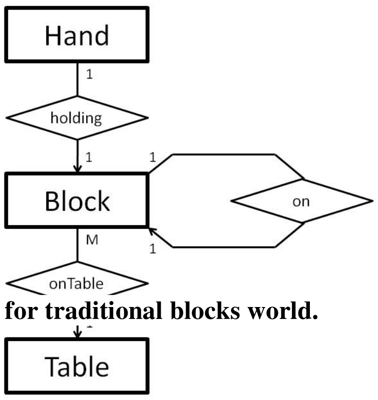

Figure 1. ERM for traditional blocks world.

block can only be held by one hand at a time. Hence, the `holding` relationship would be one (hand)-to-one (block). Figure 1 depicts the cardinality constraints as "1" and "M" (for many) at the ends of the relationships.

Changing the constraints changes the domain's behaviour. For example, if the `holding` relationship was changed to become one-to-many, then the domain's behaviour would change in that a hand could pick up a stack of blocks in a single action. The resulting domain would no longer be the traditional blocks world, but a variant of it.

A cardinality constraint is essentially a constraint between two instances of the same relationship-class. In most domains, there can also be constraints between instances from different relationship-classes. For example, in the blocks world the moment that a hand picks up a block from the table the `holding` relationship between the hand and block becomes true and the `onTable` relationship between block and table becomes false. The `holding` relationship and the `onTable` relationship are mutually excluded if they share the same block. The ERM notation is unable to depict mutual exclusion between different relationships. DIET implements Nijssen and Halpin's (1989) uniqueness (cardinality), entity type, equality, exclusion, and join constraints, but not nesting or subset constraints.

Representation

DIET takes as input a list of domain objects and a list of binary, inter-object relationships. Objects and relationships are typed. Following object-oriented design practice, we term types (a.k.a. sorts) as *classes*.

In our notation, classes are denoted by an initial capital letter (e.g. `Hand`), and instances by an initial lower-case letter and appended number (e.g. `hand12`). Variables are given the same notation as instances, prefixed with "?" (e.g. `?hand7`). Thus, Nijssen and Halpin's (1989) typing constraints are implemented implicitly.

Relationships are represented as a list, e.g. `[holding ?hand1 ?block1]`. In our ontology, relationships are binary, relating two object-instances or –variables to one another.

State is represented as a list of one or more relationships. If the relationships contain only object-instances, then the state is *literal*. For example, `[[holding hand1 block1] [onTable block2 table1]]` could represent (part of) a state of a (blocks world) domain, e.g. describing an initial or intermediate state. If one or more of the relationships in the state contain variables, then the state is *abstract*. For example, `[[holding ?hand1 ?block1] [onTable ?block2 ?table1]]` represents a set of states of the domain, e.g. (part of) a goal-state formula. By convention, if the same variable is repeated in an abstract state, then it must be instantiated by the same instance. Different variables must be instantiated by different instances. This convention implements Nijssen and Halpin's (1989) equality constraints. Thus, the abstract state `[[holding ?hand1 ?block1] [onTable ?block1 ?table1]]` would require the same block-instance to be both held and on the table simultaneously[2]. By contrast, `[[holding ?hand1 ?block1] [holding ?hand1 ?block2]]` would require different block-instances when instantiating the variables `?block1` and `?block2`.

Invariants are represented as an IF-THEN rule, e.g. `INVALID if [[holding ?hand1 ?block1] [onTable ?block1 ?table1]]`. Notice that the IF-part is a binary abstract state and the THEN-part is always the special symbol `INVALID`. A special case would be the unary invariant
`INVALID if [[on ?block1 ?block1]]`
which can be regarded as a short form for
`INVALID if [[on ?block1 ?block1] [on ?block1 ?block1]]`. Invariants implement Nijssen and Halpin's (1989) uniqueness (cardinality), exclusion, and join constraints. Two invariants are needed to express a 1-to-1 cardinality constraint.

[2] This would not be possible in the traditional blocks world. We deliberately include this example to show that our representation can also model non-traditional variants.

Domain Invariant Extraction Technique

Our technique has the following three steps:

- Step (1): Form all pairs of the relationship-classes that have at least one object-class in common. This includes pairing a relationship-class with itself.
- Step (2): Create a potential invariant for each relationship-pair, rewriting variables where necessary for the object-instances or –variables that are not in common.
- Step (3): Consult an oracle or set of traces to discover whether the potential invariant is applicable or not in the domain model.

We illustrate DIET using the traditional blocks world as described by Nilsson (1980). Nilsson's blocks world consists of a single robot hand, three blocks, and a single table. In modelling this domain, we identify three object-classes: `Hand`, `Block`, and `Table`. The corresponding instances are `hand1`, `block1`, `block2`, `block3`, and `table1`.

In Nilsson's (1980) description of the blocks world, `holding` is a relationship between the hand and one of the blocks. When the hand holds no block, it is described as being `handempty`. When a block is on the table, they are related by the `onTable` relationship. When one block is stacked on another, they are related by `on`. A block that has no block on top of it is described as being `clear`.

We represent Nilsson's (1980) `holding` relationship by `[holding hand1 ?block1]`. If `hand1` is `handempty`, then `?block1` is instantiated to `NIL`, i.e. a special symbol representing the null object. The `onTable` relationship is represented by `[onTable ?block1 table1]`. The `on` relationship is represented as `[on ?block1 ?block2]`, where `?block1` could be instantiated to `NIL` to model that the object instantiating `?block2` is `clear`.

Applying Step (1) to the Nilsson (1980) blocks world, we find that the following pairs of relationship-classes have the `Block` object-class in common:

1. `[holding hand1 ?block1]` and itself, with `?block1` in common.
2. `[holding hand1 ?block1]` and `[onTable ?block1 table1]`, with `?block1` in common.
3. `[holding hand1 ?block1]` and `[on ?block1 ?block2]`, with `?block1` in common.
4. `[holding hand1 ?block1]` and `[on ?block1 ?block2]`, with `?block1` in the `holding` relationship matched to `?block2` in the `on` relationship because they have the `Block` object-class in common.
5. `[on ?block1 ?block2]` and itself, with `?block1` in common.
6. `[on ?block1 ?block2]` and itself, with `?block2` in common.
7. `[on ?block1 ?block2]` and itself, with `?block1` in the first `on` relationship matched to

`?block2` in the second `on` relationship because they have the `Block` object-class in common.

8. The special case `[on ?block1 ?block2]`, because both `?block1` and `?block2` are variables of the common object-class `Block`.
9. `[onTable ?block1 table1]` and `[on ?block1 ?block2]`, with `?block1` in common.
10. `[onTable ?block1 table1]` and `[on ?block1 ?block2]`, with `?block1` in the `onTable` relationship matched to `?block2` in the `on` relationship because they have the `Block` object-class in common.
11. `[onTable ?block1 table1]` and itself, with `?block1` in common.

Applying Step (2), the pairings found in Step (1) would result in the corresponding potential invariants:

1. `INVALID if [[holding ?hand1 ?block1] [holding ?hand2 ?block1]]`. In essence, this invariant would state that no block can be held by multiple hands at the same time.
2. `INVALID if [[holding ?hand1 ?block1] [onTable ?block1 ?table1]]`. This invariant would state that no block can be held and on a table at the same time.
3. `INVALID if [[holding ?hand1 ?block1] [on ?block1 ?block2]]`. This invariant would state that no block can be held and stacked on another block at the same time.
4. `INVALID if [[holding ?hand1 ?block1] [on ?block2 ?block1]]`. This invariant would state that no block can be held and have another block on top of it at the same time.
5. `INVALID if [[on ?block1 ?block2] [on ?block1 ?block3]]`. This invariant would state that no block can be stacked on multiple other blocks at the same time.
6. `INVALID if [[on ?block1 ?block2] [on ?block3 ?block2]]`. This invariant would state that no block can be underneath multiple other blocks at the same time.
7. `INVALID if [[on ?block1 ?block2] [on ?block3 ?block1]]`. This invariant would state that no block can be stacked on another block and have a third block on top of it at the same time.
8. `INVALID if [[on ?block1 ?block1]]`. This invariant would state that no block can be stacked on itself.
9. `INVALID if [[onTable ?block1 ?table1] [on ?block1 ?block2]]`. This invariant would state that no block can be both on the table and stacked on another block at the same time.
10. `INVALID if [[onTable ?block1 ?table1] [on ?block2 ?block1]]`. This invariant would state that no block can be both on the table and have another block on top of it at the same time.
11. `INVALID if [[onTable ?block1 ?table1] [onTable ?block1 ?table2]]`. This invariant would state that no block can be on two (or more) tables at the same time.

Applying Step (3), an oracle – in this case Nilsson (1980) – would determine that potential invariants 7 and 10 are not applicable in the domain[3]. The output of the technique would then be the applicable invariants 1 to 6, 8, 9, and 11.

Experience shows that it is better to output all the invariants, marking each invariant as applicable or non-applicable. This makes it easier to explore the behaviour of variants of the domain by simply toggling the applicability of the invariants. For example, making invariant 5 as inapplicable would yield a variant of the traditional blocks world in which arches were allowed.

DIET has been implemented as part of a larger system for inducing STRIPS planning operators (Grant, 1996). It has been successfully tested for eight domains; see Table 1. While most are toy domains, the High Performance Capillary Electrophoresis (HPCE) domain models a standard, real-world laboratory instrument for analysing chemical samples (Eckhard, 1992).

Table 1. Domains, object- and relationship-classes.

Domain	**Objects**	**Relations**
Finger-crossing	1	1
Piano-playing	2	1
Tank-farm	2	2
Blocks world, Genesereth & Nilsson (1987) variant	2	2
Blocks world, Nilsson (1980) variant	3	3
Dining philosophers	5	5
Aircraft scheduling	9	10
HPCE laboratory instrument	24	18

The complexity of DIET is polynomial. Step (1) requires that relationship-classes be paired with one another. This indicates that the complexity is of the order of the square of the number of input relationship-classes. Assuming that, in an extreme case, there could be one relationship-class for each pair of object-classes, then the worst-case complexity would be of the order of the fourth power of the number of object-classes. However, object-classes are not fully connected by relationships in real domains, as Table 1 shows. This benign complexity behaviour makes it

[3] If they were, it would not be permissible to construct stacks of blocks in this domain.
We assume that the oracle does not decide that invariants 1 and 11 are unnecessary in a single-hand, single-table domain.

feasible to enumerate invariants exhaustively for real-world domains, such as HPCE.

Note that DIET is limited by the correctness and completeness of the input information it is given. If an object-class or a relationship is omitted, then DIET will output a subset of the domain invariants. If the oracle incorrectly marks a potential invariant as applicable or inapplicable, then DIET will deliver a set of invariants that describe a variant of the domain desired.

The example of the Nilsson (1980) blocks world illustrates the limitations caused by imperfections in the input information. In Nilsson's blocks world there is just one robot hand and one table. DIET has correctly identified the invariant that does not allow a block to be held by two (or more) robot hands (invariant 1). However, it has not identified the corresponding invariant that would disallow a hand from holding multiple blocks. Similarly, it has identified the invariant that does not allow a block to be on two (or more) tables (invariant 11). By symmetry, DIET has failed to identify a potential invariant that would disallow multiple blocks from being placed on the same table. Although the oracle would rule such an invariant as non-applicable, DIET should still generate it. Given the relationships `[holding ?hand1 ?block1]`(instead of `[holding hand1 ?block1]`) and `[onTable ?block1 ?table1]` (instead of `[onTable ?block1 table1]`), DIET generates the additional two invariants.

A major limitation inherent to DIET is that it is currently restricted to binary relationships and binary constraints. It might seem that this would only affect the more complex domains. However, these restrictions are already apparent in domains as simple as the blocks world. Intuitively, one might expect in the simplest, one-block Nilsson (1980) world that there would be just two possible states: either the block is on the table or the block is held by the robot hand. Testing yielded a third state: one in which the block is floating in mid-air, i.e. it is neither held nor resting on the table. Inspection showed that there was no invariant that modelled the action of gravity, i.e. one that required blocks either to be held by a robot hand or to be supported by another block or the table. This would need the triple invariant `INVALID if [[holding NIL ?block1] [on ?block1 NIL] [onTable ?block1 NIL]]`.

Careful thought showed that extending DIET to higher arities could lead to an infinite regress. Once triple relationships and triple invariants had been mastered, a domain would arise in which quadruple relationships and invariants were needed, then quintuple relationships and invariants, and so on.

The solution was sought elsewhere. In the object-oriented literature there are several types of relationship between object-classes. Currently, the technique exploits only one of these: instance relationships. Other notable object-oriented relationship-types are *inheritance* and *aggregation*. Research to date has concentrated on inheritance.

The approach we are working on is to change the way in which a domain is modelled by exploiting inheritance so that binary relationships and binary invariants suffice. Once again, the blocks world will illustrate our line of thinking. A clue can be seen in the similarity of the names of the `on` and `onTable` relationships. In essence, blocks and tables share the facility for supporting (stacks of) blocks above them. This common facility can be modelled by identifying a superclass – called `Support`, say – from which the `Block` and `Table` object-classes inherit; see Figure 2. The relationships and invariants expressing the common facility are linked to this `Support` superclass. By contrast, the difference between blocks and tables is that blocks can have other blocks beneath them, but tables cannot. The relationships and invariants expressing these differences are linked as appropriate either to the `Block` object-class or to the `Table` object-class. For example, an invariant may be applicable to `Block` but inapplicable to `Table`, as shown in Figure 2 for the cardinality of the `on` relationship.

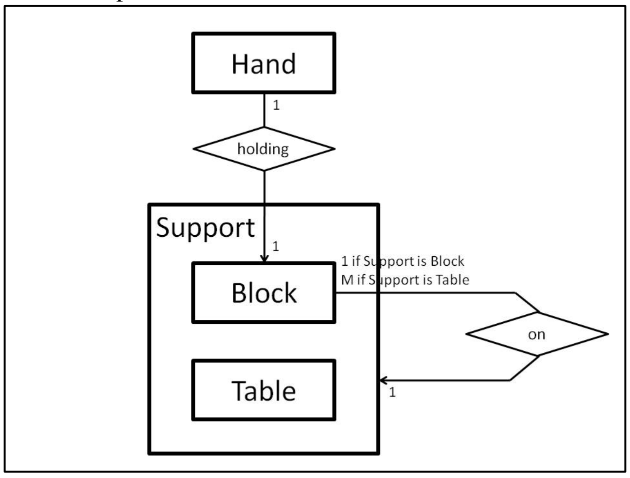

Figure 2. ERM for blocks world with inheritance.

To date, some analysis has been done by hand using inheritance to model the Nilsson (1980) blocks world and the HPCE domain. The initial results are promising. However, a modified version of DIET has yet to be formalised and implemented. Moreover, research is needed into (1) how to detect the need to change a domain model and (2) how best to change the domain model once the need for change has been detected.

Applications

Potential applications for DIET can be identified by considering the nature of its inputs. The outputs are simply sets of invariants, possibly marked as being applicable or inapplicable.

We distinguish two groups of input: (1) domain objects and their relationships, and (2) information on the applicability of the invariants generated. The first group of inputs can be obtained from one or more domain experts, from design documentation, or from observations of the real-world domain. The experts could be domain designers, planners, or instructors who train users in using the domain. The best source of information on the second group of inputs is the real-world domain itself, if it exists and is readily accessible for interrogation. The next-best source is likely to be experts who operate or control the domain. Planners and instructors may also be useful sources if they have had experience of operating or controlling the domain. The domain must exist in the real world for operators, controllers, planners, and instructors to have gained this experience. Designers are unlikely to be good sources, because, while they will be fully aware of "placard" invariants (e.g. maximum speed for flying with the undercarriage down is 273 knots), they will have little or no knowledge of operational invariants (e.g. for maximum endurance fly at Mach 0.82).

Information on the applicability of the invariants could also be obtained by extraction from traces of the operation of the domain. The trace information must be state-based, e.g. obtained by logging data produced by sensors or instrumentation attached to or embedded in the domain. If the trace includes action-based information (e.g. control system or user commands), this cannot be exploited by the technique presented in this paper[4].

Traces have limitations as a source of input information. Firstly, for traces to be obtainable at all the domain must exist in the real world. It is not possible to obtain traces for a domain that is still on the drawing board. Secondly, if we ignore operating failures, traces only provide examples of valid operating states. These examples can be used to show that an invariant hitherto considered applicable should be marked inapplicable. However, they cannot be used to change the applicability of an invariant from inapplicable to applicable. Thirdly, traces tend to be incomplete. It may be necessary to wait a long time before all possible valid states of the domain are found in the trace information.

It might seem obvious that domain experts should be able to provide the domain invariants directly, negating the need for the technique described in this paper. However, there are several reasons for claiming the contrary. Firstly, experts' knowledge is largely situational and based on personal experience (Klein, 1998). If they have not encountered a particular invariant in the past, then they may not be aware of its existence. Secondly, to operate at an expert level their knowledge must be predominantly tacit (Polanyi, 1966). This means that they can act competently using the knowledge, but have great difficulty in making the knowledge explicit to pass it on to someone else[5]. Thirdly, experts are apt to leave unstated knowledge that they regard as obvious or which "everyone knows". For example, the need to avoid reaching a state in which an explosive mixture is present in an oil refinery vessel may be so obvious to an expert that it is never mentioned. Fourthly, experts tend to operate their domain well away from invalid states, routinely remaining within a familiar, safe subset of the set of all possible valid states. In so doing, they avoid encountering many invariants. For all these reasons, experts may well be able to describe the domain objects and relationships systematically, but not the invariants.

For DIET to be useful, the following situation must hold good:

- The input source(s) must be unable to provide the invariants directly.
- Information on the applicability of invariants must be state-based.

Beyond the possible application of DIET for scientific purposes within the planning and scheduling community, there are other, wider applications that meet these criteria. All involve the specification or synthesis of a domain model. Examples include:

- Designing complex systems.
- Developing plans and operating procedures.
- Validating command-sequences during operation.
- Detecting anomalies during operation.
- Analysing commercial and military intelligence.
- Preparing examples for teaching purposes.

The designer of a complex system will be given some of the domain invariants *a priori*, e.g. the laws of nature and the "placard" invariants that the complex system must be capable of (i.e. the system specification or requirements). Others will emerge as a result of the design process, e.g. as the designer makes choices in decomposing the complex system into sub-systems and components. While the system is still on the drawing board, it may not be obvious what additional invariants have emerged as a result of the designer's choices. Moreover, there is no guarantee that the as-designed system does indeed achieve the "placard" invariants. Most design of complex systems is performed using Computer-Aided Design (CAD) software, often supporting animation of the system being designed. We envisage the technique described in this paper as obtaining its inputs from the CAD software, and returning the invariants as a way of describing the boundaries of what the as-designed system is capable of (its *operating envelope*) for animation purposes. If the operating envelope is not what the designer intended, then he/she could modify the design accordingly.

Developing plans and operating procedures (a.k.a. recipes, scripts, checklists) is a laborious process requiring expert operational knowledge. The resulting plans and procedures must not be invalid, i.e. they must not lead the user into taking actions or an action-sequence that will attempt to violate domain invariants. However, the developer may not have the operational knowledge needed, especially if the plan or procedure is for a domain that does not yet exist in the real world. DIET could support plan generation and procedure development by warning the

[4] Other techniques (e.g. Stefik, 1981; Chapman, 1987) may be applicable.

[5] Knowledge transfer from expert to novice occurs by apprenticeship or "sitting by Nellie".

developer whenever the draft plan or procedure would take the domain outside its operating envelope (Grant, 1999).

Using DIET for command-sequence validation is analogous to its use in developing operating procedures, with the difference that it supports the operators or controllers of the real-world domain rather than procedure developers. This application is already in operational use, e.g. at the European Spacecraft Operations Centre (ESOC), Darmstadt, Germany, albeit not using DIET. At ESOC, when a spacecraft controller intends to upload a command-sequence for execution by a spacecraft, he/she first executes the commands in a (ground-based) spacecraft simulator. The simulator incorporates all known invariants in the simulated spacecraft systems. The command-sequence can only be uploaded to the real spacecraft if it executes successfully in the simulator without violating any invariant. By using DIET, the invariants implemented in the spacecraft simulator could be generated automatically, rather than manually as at present.

Anomaly detection is a variant of command-sequence validation, also supporting the operators or controllers of a real-world domain. In command-sequence validation, the command sequence is executed in a simulator prior to execution in the real domain. In anomaly detection, the command sequence - after validation – is executed in the simulator and real domain in parallel. At each step in the sequence, the simulated and real states are compared. If there is a discrepancy, then execution is immediately put on hold while the anomaly is investigated. The cause could be either a failure in the real domain or faulty modelling of the domain in the simulator. One possible form of faulty modelling is an invariant that is applicable in the simulator, but inapplicable in reality, or vice versa. This application is also in operational use at ESOC, again not using DIET. Incorporating DIET would bring the same advantage as for command-sequence validation.

Analysis of commercial and military intelligence involves making sense of diverse and disjointed pieces of information about a domain. If the domain is familiar to the analyst, then analysis involves matching the information to patterns that are drawn from previous experience (Klein, 1998). Analysis of a novel domain is more difficult, and is known as *sense-making* (Dervin, 1992) (Weick, 1995). Since the domain is owned by a potential commercial or military competitor, the analyst is most unlikely to have operational experience of using it or to have access to traces of its operation. Information about the novel domain is likely to be in the form of descriptive or sensory snapshots obtained at sporadic intervals. However, the main objects making up the domain, together with their relationships, should be identifiable from these snapshots. DIET can then be used to systematically identify the potential invariants for the domain, and the analyst must use his/her knowledge of comparable domains to judge whether these invariants are applicable or not. Grant (2005) has proposed a sense-making algorithm embodying this technique. When multiple agents cooperate to pool their observations of a domain, one agent will need to assimilate the knowledge gathered by others; see (Grant, 2007).

One of the skills of an experienced teacher is to prepare a series of positive and negative examples that show how pupils may validly perform some operation, e.g. using a pen, writing letters of the alphabet, adding or subtracting numbers, and so on. The initial examples are likely to be positive ones, followed by negative examples to outline the operating envelope. These may be followed by further positive and negative examples to illustrate exceptions and to refine the operating envelope's boundaries. For example, a child learning to write might be shown how to use a pencil (positive example), but then instructed that writing on the wall or on furniture is inappropriate (negative examples). The wise teacher provides the child with amply quantities of blank paper on which to practice writing and drawing (positive examples). DIET could be used to guide the preparation of a series of examples by marking the majority of invariants as inapplicable initially as a positive stimulus, and then gradually making the appropriate invariants applicable (i.e. introducing negative examples) to delineate the true operating envelope.

Conclusions & further work

Many planning systems make use of domain constraints. Constraints may be used to speed up plan generation, to extract optimal parallel plans, or to convert a planning problem into a satisfiability equivalent. Several researchers have developed algorithms for discovering or synthesizing constraints, mostly from planning operators. This paper presents the Domain Invariant Extraction Technique (DIET) for identifying invariants – constraints that must remain true throughout the execution of a plan from initial to goal state – systematically from domain objects and their instance relationships. Unlike other knowledge engineering tools that are aimed at speeding up domain modelling, DIET aims at eliminating one possible source of human error in the modelling process, namely incompletely identifying the set of domain invariants. Drawing on the database design literature, cardinality, entity type, equality, exclusion, and join constraints are implemented, but not nesting or subset constraints. DIET's worst-case complexity is the fourth power of the number of object-classes in the domain. This benign behaviour makes it feasible to use DIET for real-world domains, such as the HPCE laboratory instrument (Eckhard, 1992).

The paper shows how a domain can be represented, describes the three-step technique, and illustrates its use by means of the blocks world. Possible applications are described in detail. This paper makes two contributions. Firstly, we add a fourth usage of constraints: to aid in the specification or synthesis of a domain model. Secondly, our technique derives invariants from the domain objects and their inter-relationships.

DIET's limitations are that relationships and invariants must be binary and invariants are only state-based. Restricting the representation to objects and relationships is not a limitation, because Chen (1976) noted that an object-attribute model can be translated into an equivalent object-relationship one, and vice versa.

In on-going further work, research is being conducted into extending DIET by exploiting the additional relationship-types of inheritance and aggregation to handle higher-arity relationships and invariants. Early results are promising.

References

Blum, A.L., & Furst, M.L. 1997. *Artificial Intelligence*, 90, 281-300.

Bresina, J., Drummond, M. & Kedar, S. 1993. Reactive, Integrated Systems Pose New Problems for Machine Learning. In Minton, S. ed. 1993. *Machine Learning Methods for Planning*. San Mateo, CA: Morgan Kaufman Publishers, 159-195 (chapter 6).

Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence*, 32, 3, 333-377.

Chen, P. P.-S. 1976. The Entity-Relationship Model - Towards a unified view of data. *ACM Transactions on Database Systems*, 1, 1, (March) 9-36.

Dean, T., Firby, J. & Miller, D. 1988. Hierarchical Planning involving Deadlines, Travel Time, and Resources. *Computational Intelligence*, 4, 4, 381-398.

Dervin, B. 1992. From the Mind's Eye of the "User": The sense-making qualitative-quantitative methodology. In Glazier, J.D., & Powell, R.R. 1992. *Qualitative Research in Information Management*. Libraries Unlimited, Englewood, CO, USA, 61-84.

Eckhard, F. 1992. High Performance Capillary Electrophoresis in the Microgravity Environment. *Advanced Space Research*, 12, 247-255.

Drummond, M.E. 1987. A Representation of Action and Belief for Automatic Planning Systems. In *Proceedings, 1986 workshop*, Los Altos, CA: Morgan Kaufman Publishing, 189-211.

Drummond, M.E. & Currie, K. 1989. *Goal-Ordering in Partially-Ordered Plans*. In Proceedings, 8th International Joint Conference on Artificial Intelligence, 960-965.

Fox, M. & Long, D. 1998. The Automatic Inference of State Invariants in TIM. *JAIR*, 9, 367-421.

Fox, M.S. 1983. *Constraint-Directed Search: A case study of job-shop scheduling*. PhD dissertation, CMU-RI-TR-83-22/CMU-CS-83-161, Pittsburgh, PA: Carnegie-Mellon University.

Genesereth, M.P. & Nilsson, N.J. 1987. *Logical Foundations of Artificial Intelligence*. Morgan Kaufman Publishers, Inc., Palo Alto, CA, USA.

Georgeff, M.P. 1987. Planning. *Annual Reviews in Computing Science*, 2, 359-400.

Gerevini, A., & Schubert, L.K. 1998. Inferring State Constraints for Domain-Independent Planning. *Proceedings, AAAI-98*, 905-912.

Grant, T.J. 1995. Generating Plans from a Domain Model. In *Proceedings, 14th workshop of UK Planning & Scheduling Special Interest Group*, Colchester, UK.

Grant, T.J. 1996. *Inductive Learning of Knowledge-Based Planning Operators*. PhD thesis, University of Maastricht, Netherlands.

Grant, T.J. 1999. The Relationship between Plans and Procedures. *Proceedings, 18th workshop of UK Planning & Scheduling Special Interest Group*, Salford, UK.

Grant, T.J. 2005. Integrating Sensemaking and Response using Planning Operator Induction. *Proceedings, 2nd international conference on Information Systems for Crisis Response And Management (ISCRAM)*, 89-96.

Grant, T.J. 2007. Assimilating Planning Domain Knowledge from Other Agents. *Proceedings, 26th workshop of UK Planning & Scheduling Special Interest Group*, 44-51.

Kautz, H., & Selman, B. 1998. The Role of Domain-Specific Knowledge in the Planning as Satisfiability Framework. *Proceedings, AIPS 1998*, 181-189.

Klein, G. 1998. *Sources of Power: How people make decisions*. MIT Press, Cambridge, MA, USA.

Kuter, U., Levine, G., Green, D., Rebguns, A., Spears, D., & DeJong, G. 2007. Learning Constraints via Demonstration for Safe Planning. *Proceedings, AAAI07*.

Lin, F. 2004. Discovering State Invariants. In *Proceedings, Knowledge Representation 2004*, paper 056.

McCluskey, T.L., & Porteous, J. 1997. Engineering and Compiling Planning Domain Models to Promote Validity and Efficiency. *Artificial Intelligence*, 95, 1-65.

Mukherji, P., & Schubert, L.K. 2005. Discovering Planning Invariants as Anomalies in State Descriptions. *Proceedings, ICAPS05*, 223-230.

Nijssen, G.M., & Halpin, T.A. 1989. *Conceptual Schema and Relational Database Design: A fact-oriented approach*. Prentice-Hall, Sydney, Australia.

Nilsson, N.J. 1980. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, USA.

Polanyi, M. 1966. *The Tacit Dimension*. Routledge & Kegan Paul, London, UK.

Refanidis, I., & Sekallarion, I. 2009. A Systematic and Complete Algorithm to Computer Higher Order Execution Relations. *Proceedings, ICAPS09, Workshop 2* (Constraint Satisfaction Techniques for Planning and Scheduling Problems), 33-42.

Rintanen, J. 2000. An Iterative Algorithm for Synthesizing Invariants. *Proceedings, AAAI-00*.

Stefik, M.J. 1981. Planning with Constraints. *Artificial Intelligence*, 16, 111-140.

Tenenburg, J.D. 1991. Abstractions in Planning. In Allen, J.F., Kautz, H.A., Pelavin, R.N., and Tenenberg, J.D. (eds.) 1991. *Reasoning about Plans*. San Mateo, CA: Morgan Kaufman Publishing.

Weick, K.E. 1995. *Sensemaking in Organizations*. Sage, Thousand Oaks, CA, USA

Zhang, Y., & Foo, N.Y. 1997. Deriving invariants and constraints from action theories. Fundamenta Informaticae, 30, 109-123.