System Assistance in Structured Domain Model Development*

Susanne Biundo and Werner Stephan
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
D-66123 Saarbriicken, Germany
{biundo, stephan}@dfki.uni-sb.de

Abstract

In this paper, we introduce a domain modeling tool that supports users in the incremental and modular development of verified models of planning domains. It relies on a logic-based concept for systematic domain model construction that provides well-defined, safe operations for the union, extension, and refinement of already existing models. The system is equiped with a deductive component. It automatically performs the proofs necessary to guarantee both the consistency of single models and the safety of operations on models. By means of detailed examples, it is shown how the system has been used for the structured development of a model for a complex, safety-critical planning domain.

1 Introduction

As soon as we aim at using planning systems in the context of realistic applications, the task of generating the underlying domain model becomes increasingly crucial. It is not only difficult to overlook the great amount of object types, relations, and actions involved when specifying such a model. It is also difficult to keep consistency in mind, which is of particular importance if the application domain is a *safety-critical* one. Consequently, system assistance in constructing safe models of complex planning domains would be of real help.

Our work is motivated by the experiment of using a deductive planning system in the simulation of a safety-critical application, namely a chemical warehouse. This planning domain is quite complex and it has turned out that the construction of a clearly structured, adequate, and consistent model would have been almost infeasible if done without any system support. The reason is that specifying a complex planning domain is an iterative process which includes frequent modifications of already specified parts and attempts to extend parts and finally put them together.

In this paper, we introduce a *domain modeling tool* (DMT) that supports users in the incremental and modular development of safe, i.e. *provably consistent,* models of planning domains. It relies on the concept of systematic domain model construction proposed by [Biundo and Stephan, 1996]. This approach—based on a modal temporal logic—considers domain models as formal structures. Domain models consist of signatures, action definitions, and domain constraints. The well-defined operations of union, extension, and refinement allow for building complex domain models out of already existing simpler ones. We have improved this domain modeling concept in order to make it really useful for practical applications. We have introduced the notion of *static* constraints, in addition to domain constraints and have extended the operations on models. These improvements grew out of a large case study, which - in a somewhat simplified version- is used as the example domain throughout this paper. Based on the modified concept, we have implemented the DMT system that supports the process of domain modeling. It does syntactic analyses of domain specifications given by a user and has a deductive component which automatically performs the proofs necessary to guarantee both the consistency of single models and the safety of the union, extension and refinement operations. Finally, we have used the system for modeling a complex, safety-critical planning domain.

The paper is organized as follows. In Section 2 we give a short introduction into the basic temporal logic formalism on which the domain modeling process relies. Our formal concept of domain model development and the domain modeling tool that implements this concept are presented in Section 3. We give an overview of the chemical warehouse planning domain CHEW in Section 4 and demonstrate the structured development of a CHEW domain model by means of examples in Section 5.

"This work has partly been supported by the Federal Ministry of Education, Science, Research, and Technology under grants ITW 9404 and ITW 9600.

2 The Logical Setting

The logical representation formalism is a subset of the temporal planning logic TPL introduced in [Stephan and Biundo, 1996]. We extend this formalism by the concept of nondeterministic (*random*) assignments, but do not consider recursive actions or plans. TPL is an interval-based modal temporal logic. It relies on a many-sorted first-order language that provides *flexible* and *rigid* function and relation symbols. The modal operators include $\bigcirc$ (weak next), $\odot$ (strong next), $\Box$ (always), $\Diamond$ (sometimes), and the binary operator ; (chop). TPL formulae are interpreted over *intervals*. Intervals $\bar{\sigma}$ are nonempty sequences of states ($\bar{\sigma}=<\sigma_1, \sigma_2 \ldots>$). States σ_i are interpretations of the flexible symbols. A first-order formula ϕ holds in an interval $\bar{\sigma}$ iff it holds in the first state of that interval. $\bigcirc\varphi$ holds in $\bar{\sigma}$ iff $\bar{\sigma}$ is a single-state interval ($\bar{\sigma}=<\sigma_1>$) or φ holds in the greatest proper suffix of $\bar{\sigma}$ (which is $<\sigma_2, \ldots>$). $\bigcirc$ **false** means that there exists no such suffix. An interval $\bar{\sigma}$ satisfies $\odot\varphi$ iff $\bar{\sigma}$ is *not* a single-state interval and satisfies $\bigcirc\varphi$, i.e., $\bar{\sigma}$ has a greatest proper suffix that satisfies φ. $\Box\varphi$ and $\Diamond\varphi$ hold in $\bar{\sigma}$ iff φ holds in every suffix of $\bar{\sigma}$ and iff there exists a suffix of $\bar{\sigma}$ that satisfies φ, respectively. The chop-operator is interpreted as the *sequential composition* of formulae: φ ; ψ holds in $\bar{\sigma}=<\sigma_1, \sigma_2, \ldots, \sigma_n, \ldots>$ iff either $\bar{\sigma}$ is infinite and satisfies φ or $\bar{\sigma}$ can be split into two successive intervals $\bar{\sigma}'=<\sigma_1, \sigma_2, \ldots, \sigma_n>$ and $\bar{\sigma}''=<\sigma_n, \ldots>$—which overlap in one state—such that $\bar{\sigma}'$ satisfies φ and $\bar{\sigma}''$ satisfies ψ.

In addition to these modalities, a *programming language* is embedded in TPL. This programming language is used to specify the effects of basic actions. It provides elementary operations **add-**.., **delete-**.., and := (update), which represent single-step transitions described by $\bigcirc$. As for control structures, ; is used as sequential composition and the following axioms are given.

$$\textbf{skip} :\leftrightarrow \bigcirc\ \textbf{false} \tag{1}$$

$$\textbf{if}\ \phi\ \textbf{then}\ \pi_1\ \textbf{else}\ \pi_2\ \textbf{fi} :\leftrightarrow ((\phi \rightarrow \pi_1) \wedge (\neg\phi \rightarrow \pi_2)) \tag{2}$$

$$\textbf{choose}\ x : \phi\ \textbf{begin}\ \pi\ \textbf{end} :\leftrightarrow ((\exists x\ (\phi \wedge \pi)) \vee (\neg\exists x\ \phi \wedge \textbf{skip})) \tag{3}$$

Actions are defined by expressions like $a(x_1, \ldots, x_n) \Leftarrow \pi$, where 'a' is the action name, $(x_1, \ldots, x_n)$ are the parameters, and π is a program. Programs are built using elementary operations, *action calls* $a(x_1, \ldots, x_n) \Leftarrow \pi$ $(t_1, \ldots, t_n)$, and control structures. Programs are TPL formulae. In general, these programs are used to describe not only actions but also composite plans.

The elementary operations are characterized by axioms[1], like

$$(\textbf{add-}r(t_1, \ldots, t_n) \quad \wedge\ \tilde{\phi}) \quad \rightarrow \quad (\odot\bigcirc\ \textbf{false} \wedge \bigcirc\phi)$$

$$(f(x_1, \ldots, x_n) :=?: z \wedge \dot{\phi}) \rightarrow$$
$$(\odot\bigcirc\ \textbf{false} \wedge \bigcirc\phi \wedge \bigcirc(\exists x : z\ f(x_1, \ldots, x_n) = x))$$

where r (f) is a flexible relation (function) symbol and $x : z$ is a fresh variable of type z. $f(\ldots) :=?: z$ represents the random assignment. $\tilde{\phi}$ is the formula resulting from the first-order formula ϕ by replacing each occurrence of the atomic subformula $r(s_1, \ldots, s_n)$ by $((t_1 \neq s_1 \vee \ldots \vee t_n \neq s_n) \rightarrow r(s_1, \ldots, s_n))$ and $\dot{\phi}$ results from ϕ by replacing $f(s_1, \ldots, s_n) = s$ by $((f(s_1, \ldots, s_n) = s \wedge (x_1 \neq s_1 \vee \ldots \vee x_n \neq s_n))$.
$\tilde{\phi}$ and $\dot{\phi}$ are the weakest preconditions of ϕ w.r.t. the respective elementary operation[2]. Please note that elementary operations describe single state transitions, whereas actions and plans are composed of many of them using the chop-operator, thereby specifying intervals of arbitrary length.
Finally, for action calls we have the equivalence[3]

$$a(\bar{x}) \Leftarrow \pi(\bar{t}) \quad \leftrightarrow \quad \pi[\bar{x}/\bar{t}][a/a(\bar{x}) \Leftarrow \pi]. \tag{4}$$

3 The Domain Modeling Tool

In [Biundo and Stephan, 1996], we proposed a formal concept for the construction of *provably consistent* planning domain models. It provides well-defined operations on models that allow for the structured and incremental development of domain models by combining and extending already existing ones in a sound manner. We extend this concept by introducing the notion of *static constraints* and by extending the *refinement* operation in a way such that also abstract *relations* can be "implemented" by concreter ones.
A domain model M is a data structure <Z, Sig, SC, DC, A>, where Z is a set of sort symbols, Sig is a signature, SC and DC are sets of static constraints and domain constraints, and A is a set of action definitions. SC and DC consist of first-order formulae, whereby only rigid symbols are allowed in SC. Static constraints serve to describe *unchangeable* domain features, whereas domain constraints reflect essential *changeable* facts that nevertheless have to hold in each situation. Domain constraints therefore contain flexible symbols. A domain model is consistent if it is syntactically sound (i.e. symbols are used in a unique way and terms and formulae are well-formed) and if each domain constraint is invariant against each action (i.e. it holds after the execution of an action iff it has held before). Please note that the latter trivially holds for static constraints.
Operations on domain models are *union, extension,* and *refinement* The operations are safe. They guarantee that only consistent models develop, given consistent ones as input.

[1] We refer to these as the "wp-axioms".

[2] The wp-axioms for ordinary assignments and the delete operation are omitted, due to lack of space.

[3] $\bar{x}$ ($\bar{t}$) denotes a vector of variables (terms).

Union ($\uplus$) combines two domain models M_1 and M_2, provided:

- Z_1, Sig_1 and Z_2, Sig_2 agree on common sorts, i.e.: common sorts are identical w.r.t. subsort and supersort relationships and the signatures match w.r.t. symbols ranging over common sorts.
- Actions from $A_1 \cup A_2$ that use function and relation symbols from both Sig_1 and Sig_2 are in $A_1 \cap A_2$.
- Formulae from $SC_1 \cup SC_2$ ($DC_1 \cup DC_2$) that use function and relation symbols from both Sig_1 and Sig_2 are in $SC_1 \cap SC_2$ ($DC_1 \cap DC_2$).

$M_1 \uplus M_2 := <Z_1 \cup Z_2,\ Sig_1 \oplus Sig_2,\ SC_1 \cup SC_2,\ DC_1 \cup DC_2,\ A_1 \cup A_2>$, where $\oplus$ denotes the union of signatures. The consistency of $M_1 \uplus M_2$ is given by the consistency of M_1 and M_2. In particular, the invariance of DC_1-DC_2 (DC_2-DC_1) against the actions A_2 (A_1) is guaranteed as these actions and formulae have no symbols in common.

Extensions ($\triangle$) add new sorts, symbols, actions, or constraints to a model $M = <Z, Sig, SC, DC, A>$. Let $E = <Z_e, Sig_e, SC_e, DC_e, A_e>$. M can be extended by E if:

- Z and Z_e agree on common sorts.
- Sig and Sig_e have no symbols in common.
- A_e are action definitions over $Sig \oplus Sig_e$.
- SC_e and DC_e are constraints over $Sig \oplus Sig_e$.

$M \triangle E := <Z \cup Z_e, Sig \oplus Sig_e, SC \cup SC_e, DC \cup DC_e, A' \cup A_e>$, where A' results from A by extending (some) action definitions by parts that change only symbols from Sig_e and also by perhaps restricting their applicability. In order to guarantee consistency of $M \triangle E$ (given that M and E are consistent), it has to be proved that the actions A' (A_e) are invariant against DC_e ($DC \cup DC_e$).

Refinements ($\triangledown$) allow for the replacement of abstract scenarios by more concrete ones. Let M be given. In a first step M is extended by some E yielding $M \triangle E$, like above. In particular this means that some of the previous actions a are extended to a'. In addition to that there are completely new actions using only symbols from Sig_e.

In a second step we want to *forget* about some of the previous symbols and actions. In a sense these symbols and also some of the actions using them will be *implemented* in terms of new concepts which in turn means that facts about the abstract scenario are preserved in a somewhat modified form. Let the symbols we want to implement (and then forget about) be given by Sig_f and let $RP = \{\forall \bar{x}(r(\bar{x}) \leftrightarrow \phi_r) \mid r \in Sig_f\} \cup \{\forall \bar{x}\bar{y}(f(\bar{x}) = \bar{y} \leftrightarrow \phi_f) \mid f \in Sig_f\}$, where the ϕ_r (ϕ_f) are first-order formulae not containing symbols from Sig_f. The formulae in RP have to be proved to be invariants of the actions in $M \triangle E$. However, some of the actions in A' shall no longer be available on the concrete level, i.e. in the refined model. These are given by $A_f \subseteq A'$. We then replace atomic formulae containing the symbols from Sig_f in each constraint ϕ of $M \triangle E$ by the equivalences given in RP. This can be done by a straightforward algorithm. Let ϕ^* be the formula resulting from ϕ after this step. For an action a the transformed version a^* is obtained by translating the tests in the way described above and by removing all elementary operations that affect symbols from Sig/. If function symbols from Sig/ occur in arguments of the remaining elementary operations, these have to be eliminated by using the choose construct. We end up with a set of actions $(A'-A_f)^*$. The refinement finally is $(M \triangle E) \triangledown <Sig_f, RP, A_f> := <Z \cup Z_e, (Sig \oplus Sig_e)\text{-}Sig_f, (SC \cup SC_e)^*, (DC \cup DC_e)^*, (A'-A_f)^* \cup A_e>$

Based on this formal concept of domain model construction, we have implemented a deduction-based *domain modeling tool* (DMT) that supports the modeling process by carefully analyzing an user's inputs and by automatically performing the necessary proofs. The system has components for both syntactic analysis and deduction. Syntactic analysis guarantees well-formedness of terms and formulae and proves that no flexible symbols occur in static constraints and that no rigid symbols are manipulated by update-, add-, or delete-operations. In order to prevent the user from specifying an action that causes undesired effects or no effects at all, the system does the following analysis in addition. It checks the precondition and body of each action. If it detects flexible terms in the precondition that are identical with terms being manipulated in the action body, it tries to prove that the values they have before and after the action execution differ. This is done by the deductive component of DMT which tries to derive this fact from the preconditions, using the static constraints and the domain constraints. The deductive component is an automated theorem prover for TPL.

4 The Chemical Warehouse Scenario

The chemical exchange warehouse (CHEW) was developed at the *Hazardous Waste Management Division* (HWM) of the Lawrence Livermore National Lab[4]. The CHEW concept is part of a programme for the environmentally sound, cost effective and legal management of chemical waste. It is devoted to the storage of excess usable chemicals, the resale of inquired substances, and the disposal of unsaleable ones. Dealing with dangerous materials, the chemical warehouse is a safety-critical environment. The HWM concept takes this fact into account by specifying a collection of *safety conditions* that must not be violated by anyone who acts in this environment. We have implemented a simulation of the CHEW warehouse (cf. Figure 1) where—in contrast to what the HWM concept says—we assume that a robot is used for the transportation of barrels. Barrels containing

[4] http://www.llnl.gov/es_andJh/hwmJiome/hwm.html

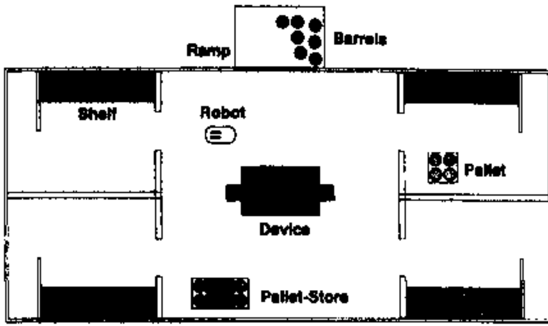

Figure 1: The Chemical Exchange Warehouse

chemicals are delivered at the ramp. First, the barrels are proved and their content is analyzed. The barrels are then stored in suitable cells from where they can be retrieved as soon as respective inquiries have been submitted. A deductive planning system is used to generate plans that enable the robot to act safely in this (simulated) environment by executing *provably correct* plans that in particular meet the required safety conditions.

As a prerequisite for using our planning system, we had to set up a model of this domain. In order to obtain a suitable and consistent, comprehensive model we have constructed a *development tree* using the DMT domain modeling tool. This tree reflects how separate parts of

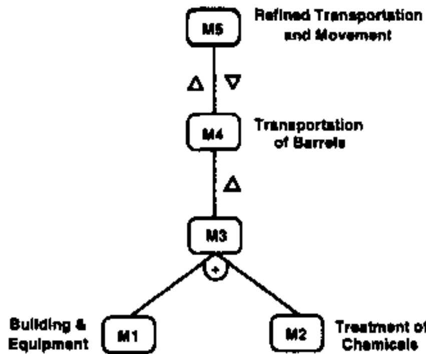

Figure 2: Development of the CHEW Domain Model

the model have been specified, flexibly extended and refined, and finally tied together. Figure 2 shows a (simplified) part of this development tree and Section 5 gives some detailed examples on the development steps.

5 A Structured Domain Model Development

We start developing the CHEW model by first specifying some basic components of the warehouse building and its equipment. There are rooms, doors, and a robot. The rooms are connected by doors and the robot is able to move from one room to another. We set up a domain model $M_1 = <Z_1, Sig_1, SC_1, DC_1, A_1>$, where $Z_1 = \{room, door, robot\}$. The signature Sig_1 consists of $R_r = \{connects_{(door,room,room)}\}$, $F_f = \{in\text{-}r_{robot \to room}\}$, $F_r = \{rob_{\to robot}\}$; $R_f = \{\}$. There is only one robot supposed to be in our scenario; it is denoted by a constant. We have two static constraints saying that a door connects two *different* rooms, and that the *connects* relation is symmetric w.r.t. the *room* arguments: $SC_1 = \{\forall d, r_1, r_2 (connects(d, r_1, r_2) \rightarrow r_1 \neq r_2),$
$\forall d, r_1, r_2 (connects(d, r_1, r_2) \leftrightarrow connects(d, r_2, r_1))\ \}$. As the TPL representation formalism allows for the use of flexible *function* symbols, we are able to represent unique relations like *in-r* by functions. This preserves us from specifying and maintaining a domain constraint explicitely stating the uniqueness of *in-r*, namely $\forall r_1, r_2 ((in\text{-}r(rob, r_1) \wedge in\text{-}r(rob, r_2)) \rightarrow r_1 = r_2)$. Finally, we define an action that allows the robot to move from one room to another: A_1 consists of

move(r_1 : *room*, r_2 : *room*) ⇐
  choose $d : connects(d, r_1, r_2)$
  begin if $in\text{-}r(rob) = r_1$ **then**
    $in\text{-}r(rob) := r_2$ **else skip fi end**

As already explained in Section 3, the DMT tries to prevent the user from specifying an action that causes undesired effects or no effects at all. In our example, the system detects a flexible term in the precondition of 'move' that is changed in the action body, namely *in-r(rob)*. Therefore, it tries to derive the unequality of the respective values from the preconditions using the static constraints: $SC \vdash \forall r_1, r_2, d\ ((connects(d, r_1, r_2) \wedge in\text{-}r(rob) = r_1) \rightarrow r_1 \neq r_2)$. The system succeeds with this proof as the first static constraint in SC states that the rooms connected by a door are different. This way, static constraints serve to keep certain (static) facts out of preconditions which helps to make action definitions simpler.

In a second step, we concentrate on the treatment of chemicals in our warehouse scenario. We specify the respective objects and, among others, an action that represents the analysis of the content of a delivered barrel. The analysis process is modeled on an abstract level, because in reality it is carried out by a human. Thus, the analysis device is considered to be a black box. In order to analyze its content a barrel has to be put in the device. After the analysis process the barrel is put out and has a color mark indicating the type of chemical it contains (cf. Figure 3).
We set up the domain model $M_2 = <Z_2, Sig_2, SC_2, DC_2, A_2>$, where $Z_2 = \{barrel, color, default.color, new.color, device\}$. Here, we have a hierarchy of sorts. *color* has two subsorts: *default.color* and *new.color*.

Figure 3: Analyzing the Content of a Barrel

Sig_2 consists of $R_f=\{in_{(barrel)}, out_{(barrel)}, clear\text{-}in_{()}, clear\text{-}out_{()}\}$, $R_r=\{\}$, $F_f=\{paint_{barrel\to color}\}$, and $F_r= \{dev_{\to device}, blue_{\to new.color}, red_{\to new.color}, green_{\to new.color}, yellow_{\to new.color}, grey_{\to default.color}\}$. As static constraints we have that different constant symbols denote different objects, also known as *unique name assumption*. In order to correctly model our planning domain, we have to make sure that the model reflects essential properties of that domain, like "a barrel cannot be located at the input- and output position of the device at the same time". Therefore, suitable domain constraints are introduced. We have $DC_2 = \{\forall b' \neg(in(b') \land out(b')), clear\text{-}in \leftrightarrow \neg\exists\, b\; in(b), clear\text{-}out \leftrightarrow \neg\exists\, b\; out(b)\}$. The analysis action is defined as follows.

analyze(b:*barrel*) $\Leftarrow$
  if *in*(*b*) $\land$ *clear-out* **then**
    delete-*in*(*b*) ; **add**-*clear-in* ; *paint*(*b*) :=? ;
    add-*out*(*b*) ; **delete**-*clear-out* **else skip fi**

As we proceed on the assumption that the content of a barrel is not known unless it has been analyzed, the action is formulated as a *nondeterministic* one. It assigns an "unknown" value ? to *paint*(*b*). As one effect of this action, we have that there exists an object of type *new.color* that equals *paint*(*b*). In addition, we have actions for loading and unloading the device. 'load', for example, reads:

load(b:*barrel*) $\Leftarrow$
  if *clear-in* **then**
    delete-*clear-in* ; **add**-*in*(*b*) **else skip fi**

During the specification of M_2, DMT provides type checks and proves syntactical soundness of terms and formulae. In addition, it has to process the domain constraints. This means, the domain constraint

$$\varphi \equiv \forall b' \neg(in(b') \land out(b'))$$

of DC_2 has to be proved invariant against 'analyze', for example. To do so, the system generates the invariance assertion [5]:
$(\varphi \land analyze(b)) \rightarrow \Diamond\, (\bigcirc\, \mathbf{false} \land \varphi)$ stating: If φ holds in a state where we execute 'analyze', then we end up in a state where φ holds again.

[5] *analyze*(*b*) abbreviates the action call analyze(b:*barrel*) $\Leftarrow \pi\ (b)$, the variable *b* being the actual parameter.

In order to prove this formula, the system first applies the axioms for control structures given in Section 2. Then it iteratively computes weakest preconditions of φ w.r.t. the elementary operations in the body of 'analyze', using the wp-axioms. As a result it obtains the first-order implication
$\forall b\, ((\forall b' \neg(in(b') \land out(b')) \land in(b)) \rightarrow$
$\forall b' \neg(in(b') \land b \neq b' \land out(b')))$.
It is easy to see, that this formula can be simplified to **true**.

In the next step, we want to combine M_1 and M_2, thereby introducing the concepts of barrels and their analyses into the model of the building. The combination of these models is achieved by applying the union operation for models. The prerequisites for this operation are obviously satisfied: The models have no sorts in common and with that differ also in signatures, actions, and constraints. We obtain $M_3 = M_1 \uplus M_2$, where $Z_3 = Z_1 \cup Z_2$, $Sig_3 = Sig_1 \oplus Sig_2$, $SC_3 = SC_1 \cup SC_2$, $DC_3 = DC_2$, and $A_3 = A_1 \cup A_2$.
Now, we extend M_3 by modeling the transportation of barrels using the robot. We define the extension $E = <Z_3, Sig, \{\}, DC, A>$. Sig has the flexible symbols: $R_f=\{held_{(barrel)}, handempty_{()}\}$ and $F_f=\{in\text{-}b_{barrel\to room}\}$. There are no static constraints, but three domain constraints: $DC = \{handempty \leftrightarrow \neg\exists\, b\; held(b)$,
$\forall b\, (held(b) \rightarrow in\text{-}r(rob) = in\text{-}b(b))$,
$\forall\, b_1, b_2\, ((held(b_1) \land held(b_2)) \rightarrow b_1 = b_2)\}$. Note that we have used a flexible *relation* symbol for the unique relation *held*, in contrast to what has been done in the cases of *in-r* and *in-b* and at the cost of needing a domain constraint explicitely stating the uniqueness. The reason is that using a flexible function symbol instead (describing the *held* property by formulae like *held*=b) would have required the introduction of a so-called error element, like *empty-barrel*, in order to express the fact that nothing is held by the robot. This in turn would have required an expensive exception handling in the body of actions that manipulate barrels, like case analyses of the form **if** $b \neq$ *empty-barrel* **then** ... **else skip**.
A contains the actions for picking up and putting down barrels. 'pickup' reads:

pickup(*b* : *barrel*) $\Leftarrow$
  if *handempty* $\land$ *in-r*(*rob*) = *in-b*(*b*) **then**
    delete-*handempty* ; **add**-*held*(*b*) **else skip fi**

The DMT syntax check detects no errors and the system also succeeds in proving the domain constraints DC invariant against 'pickup' and 'putdown'. Now we can extend M_3 by E as the prerequisites of the extension operation Δ are satisfied: The signatures are disjoint and A and DC are built over $Sig_3 \oplus$ Sig. We generate the extension $M_3\ \Delta\ E = M_4$ where $Z_4 = Z_3$, $Sig_4 = Sig_3 \oplus$ Sig, $SC_4 = SC_3$, $DC_4 = DC_3 \cup DC$, and $A_4 = A_3' \cup A$.

A_3' results from A_3 by inserting some expressions from Sig into the definition of 'move':

move(r_1 : *room*,r_2 : *room*) $\Leftarrow$
  choose d : *connects*(d, r_1, r_2)
  begin **if** *in-r*$(rob) = r_1$ **then**
    in-r$(rob) := r_2$;
    choose b : *held*(b)
      begin *in-b*$(b) := r_2$ **end**
    else **skip** **fi** **end**

This modification is necessary, in order to get the second domain constraint of DC invariant against 'move'. With that, the system succeeds in performing the proof obligations that guarantee safety of our extension, namely: DC_3 are invariants w.r.t. A ('pickup' and 'putdown' manipulate only symbols from Sig) and DC are invariants w.r.t. A_3' ('analyze', 'load', and 'unload' do not affect symbols occurring in DC). So, M_4 is accepted by the system.

Finally, we will show how our new model M_4 is *refined* to a more concrete one by introducing the notion of *location* that serves to represent the various positions inside the building at which movable objects, like barrels and the robot, could be located. This notion will enable the specification of "moving inside a room", for example.

As described in Section 3, the operation of domain model refinement comprises extension as well as the elimination of actions and symbols. Assume we first extend M_4 by some E = <Z, Sig, SC, DC, A>, where Z = {*location*} and among the rigid symbols we have $entrance_{device \rightarrow location}$, $exit_{device \rightarrow location}$, and $area_{location \rightarrow room}$. The flexible symbols include $at\text{-}r_{robot \rightarrow location}$ and $at\text{-}b_{barrel \rightarrow location}$. We add also actions for moving inside a room and for passing a door, respectively. After the extension step, we want to forget about the previous 'move' action of M_4 as well as about the relations *in* and *out*, originating from M_2. Let RP contain the formulae $\forall\ b\ (in(b) \leftrightarrow at\text{-b}(b) = entrance(dev))$ and $\forall\ b\ (out(b) \leftrightarrow at\text{-b}(b) = exit(dev))$. It turns out that 'analyze' is among the actions of A_4 that have to be modified in order to enable the invariance proof of the RP formulae: We have to consider locations here:

analyze(b:*barrel*) $\Leftarrow$
  if *in*(b) $\wedge$ *clear-out* $\wedge$
    at-b$(b) = entrance(dev)$ **then**
    delete-*in*(b) ; **add**-*clear-in* ; *paint*(b) :=? ;
    add-*out*(b) ; **delete**-*clear-out* ;
    at-b$(b) := exit(dev)$ **else** **skip** **fi**

It is easy to see that the first formula of RP still cannot be proved to be invariant against 'analyze' unless we have added the state constraint $entrance(dev) \neq exit(dev)$.

As can be seen from this example, the generation of sound models can be a tricky task, the deductive support of which is really useful. In this case an analysis of the failed invariance proof has helped us modifying the extension accordingly unless we ended up with a correctly refined domain model.

6 Related Work and Conclusion

The question of knowledge acquisition and knowledge base maintenance for planning has hardly been addressed in the literature. Recently, [Chien, 1996] has introduced an approach to the analysis of planning knowledge bases. This mechanism detects goals that are notachievable by actions and supports the user in detecting modeling errors by allowing the generation of automatically completed plans. In the work of [Cesta and Oddi, 1996], a formal domain description language has been proposed, which is especially well suited for the description of physical planning domains. This approach also considers domain model construction based on a formal semantics.

In this paper, we have introduced the domain modeling tool DMT that assists users in the modular and structured development of verified domain models. It provides well-defined operations for the extension, refinement, and combination of existing models and automatically performs the proofs that are necessary to guarantee the safety of these operations.

Although our system uses the TPL planning logic, DMT can easily be adapted—by restricting the logical language—to construct safe domain models for operator-based planning systems that rely on different formalisms; examples being systems using STRIPS-like operator descriptions or systems based on ADL, like UCPOP [Penberthy and Weld, 1992].

References

[Biundo and Stephan, 1996] S. Biundo and W. Stephan. Modeling Planning Domains Systematically. In *Proc. of ECAI-96,* pages 599-603. Wiley & Sons, 1996.

[Cesta and Oddi, 1996] A. Cesta and A. Oddi. DDL.I: A Formal Description of a Constraint Representation Language for Physical Domains. In *New Directions in AI Planning,* pages 341 352. IOS Press, 1996.

[Chien, 1996] S. A. Chien. Static and Completion Analysis for Planning - Knowledge Base Development and Verification. In *Proc. of AIPS-96,* pages 53-61. AAAI Press, 1996.

[Penberthy and Weld, 1992] J. S. Penberthy and D. S. Weld. UCPOP A Sound, Complete, Partial Order Planner for ADL. In *Proc. of KR-92,* pages 103-114, 1992.

[Stephan and Biundo, 1996] W. Stephan and S. Biundo. Deduction-Based Refinement Planning. In *Proc. of AIPS-96,* pages 213-220. AAAI Press, 1996.