Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

January 20, 2020

Contents

theory *Model-Enumeration*
imports *Entailment-Definition.Partial-Annotated-Herbrand-Interpretation Weidenbach-Book-Base.Wellfounded-More*
begin

lemma *Ex-sat-model*:
assumes ‹*satisfiable* (*set-mset* *N*)›
shows ‹∃ *M*. *set M* ⊨*sm N* ∧
distinct M ∧
consistent-interp (*set M*) ∧
atm-of ' *set M* ⊆ *atms-of-mm N*›
⟨*proof*⟩

definition *all-models* **where**
‹*all-models N* = {*M*. *set M* ⊨*sm N* ∧ *consistent-interp* (*set M*) ∧
distinct M ∧ *atm-of* ' *set M* ⊆ *atms-of-mm N*}›

lemma *finite-all-models*:
‹*finite* (*all-models N*)›
⟨*proof*⟩

inductive *next-model* **where**
‹*set M* ⊨*sm N* ⟹ *distinct M* ⟹ *consistent-interp* (*set M*) ⟹
atm-of ' *set M* ⊆ *atms-of-mm N* ⟹ *next-model M N*›

lemma *image-mset-uminus-eq-image-mset-uminus-literals*[*simp*]:
‹*image-mset uminus M'* = *image-mset uminus M* ⟷ *M* = *M'*› **for** *M* :: ‹*'v clause*›
⟨*proof*⟩

context
fixes *P* :: ‹*'v literal set* ⇒ *bool*›
begin

inductive *next-model-filtered* :: ‹*'v literal list option* × *'v literal multiset multiset*
⇒ *'v literal list option* × *'v literal multiset multiset*
⇒ *bool*› **where**
‹*next-model M N* ⟹ *P* (*set M*) ⟹ *next-model-filtered* (*None*, *N*) (*Some M*, *N*)› |
‹*next-model M N* ⟹ ¬*P* (*set M*) ⟹ *next-model-filtered* (*None*, *N*) (*None*, *add-mset* (*image-mset uminus* (*mset M*)) *N*)›

lemma *next-model-filtered-mono*:
‹*next-model-filtered a b* ⟹ *snd a* ⊆# *snd b*›
⟨*proof*⟩

lemma *rtranclp-next-model-filtered-mono*:
‹$next\text{-}model\text{-}filtered^{**}\ a\ b \Longrightarrow snd\ a \subseteq\#\ snd\ b$›
⟨*proof*⟩

lemma *next-filtered-same-atoms*:
‹$next\text{-}model\text{-}filtered\ a\ b \Longrightarrow atms\text{-}of\text{-}mm\ (snd\ b) = atms\text{-}of\text{-}mm\ (snd\ a)$›
⟨*proof*⟩

lemma *rtranclp-next-filtered-same-atoms*:
‹$next\text{-}model\text{-}filtered^{**}\ a\ b \Longrightarrow atms\text{-}of\text{-}mm\ (snd\ b) = atms\text{-}of\text{-}mm\ (snd\ a)$›
⟨*proof*⟩

lemma *next-model-filtered-next-modelD*:
‹$next\text{-}model\text{-}filtered\ a\ b \Longrightarrow M \in\#\ snd\ b - snd\ a \Longrightarrow M = image\text{-}mset\ uminus\ (mset\ M') \Longrightarrow next\text{-}model\ M'\ (snd\ a)$›
⟨*proof*⟩

lemma *rtranclp-next-model-filtered-next-modelD*:
‹$next\text{-}model\text{-}filtered^{**}\ a\ b \Longrightarrow M \in\#\ snd\ b - snd\ a \Longrightarrow M = image\text{-}mset\ uminus\ (mset\ M') \Longrightarrow next\text{-}model\ M'\ (snd\ a)$›
⟨*proof*⟩

lemma *rtranclp-next-model-filtered-next-false*:
‹$next\text{-}model\text{-}filtered^{**}\ a\ b \Longrightarrow M \in\#\ snd\ b - snd\ a \Longrightarrow M = image\text{-}mset\ uminus\ (mset\ M') \Longrightarrow \neg P\ (uminus\ `\ set\text{-}mset\ M)$›
⟨*proof*⟩

lemma *next-model-decreasing*:
assumes
‹$next\text{-}model\ M\ N$›
shows ‹$(add\text{-}mset\ (image\text{-}mset\ uminus\ (mset\ M))\ N,\ N) \in measure\ (\lambda N.\ card\ (all\text{-}models\ N))$›
⟨*proof*⟩

lemma *next-model-decreasing'*:
assumes
‹$next\text{-}model\ M\ N$›
shows ‹$((P,\ add\text{-}mset\ (image\text{-}mset\ uminus\ (mset\ M))\ N),\ P,\ N) \in measure\ (\lambda(P,\ N).\ card\ (all\text{-}models\ N))$›
⟨*proof*⟩

lemma *wf-next-model-filtered*:
‹$wf\ \{(y,\ x).\ next\text{-}model\text{-}filtered\ x\ y\}$›
⟨*proof*⟩

lemma *no-step-next-model-filtered-unsat*:
assumes ‹$no\text{-}step\ next\text{-}model\text{-}filtered\ (None,\ N)$›
shows ‹$unsatisfiable\ (set\text{-}mset\ N)$›
⟨*proof*⟩

lemma *unsat-no-step-next-model-filtered*:
assumes ‹$unsatisfiable\ (set\text{-}mset\ N)$›
shows ‹$no\text{-}step\ next\text{-}model\text{-}filtered\ (None,\ N)$›
⟨*proof*⟩

lemma *full-next-model-filtered-no-distinct-model*:
assumes
no-model: ‹*full next-model-filtered* (*None*, *N*) (*None*, *N'*)› **and**
filter-mono: ‹$\bigwedge$*M M'*. *set M* $\models$*sm N* $\Longrightarrow$ *consistent-interp* (*set M*) $\Longrightarrow$ *set M'* $\models$*sm N* $\Longrightarrow$
distinct M $\Longrightarrow$ *distinct M'* $\Longrightarrow$ *set M* $\subseteq$ *set M'* $\Longrightarrow$ *P* (*set M*) $\longleftrightarrow$ *P* (*set M'*)›
shows
‹$\nexists$*M*. *set M* $\models$*sm N* $\wedge$ *P* (*set M*) $\wedge$ *consistent-interp* (*set M*) $\wedge$ *distinct M*›
⟨*proof*⟩

lemma *full-next-model-filtered-no-model*:
assumes
no-model: ‹*full next-model-filtered* (*None*, *N*) (*None*, *N'*)› **and**
filter-mono: ‹$\bigwedge$*M M'*. *set M* $\models$*sm N* $\Longrightarrow$ *consistent-interp* (*set M*) $\Longrightarrow$ *set M'* $\models$*sm N* $\Longrightarrow$
distinct M $\Longrightarrow$ *distinct M'* $\Longrightarrow$ *set M* $\subseteq$ *set M'* $\Longrightarrow$ *P* (*set M*) $\longleftrightarrow$ *P* (*set M'*)›
shows
‹$\nexists$*M*. *set M* $\models$*sm N* $\wedge$ *P* (*set M*) $\wedge$ *consistent-interp* (*set M*)›
(**is** ‹$\nexists$*M*. *?P M*›)
⟨*proof*⟩

end

lemma *no-step-next-model-filtered-next-model-iff*:
‹*fst S* = *None* $\Longrightarrow$ *no-step* (*next-model-filtered P*) *S* $\longleftrightarrow$ ($\nexists$*M*. *next-model M* (*snd S*))›
⟨*proof*⟩

lemma *Ex-next-model-iff-statisfiable*:
‹($\exists$*M*. *next-model M N*) $\longleftrightarrow$ *satisfiable* (*set-mset N*)›
⟨*proof*⟩

lemma *unsat-no-step-next-model-filtered'*:
assumes ‹*unsatisfiable* (*set-mset* (*snd S*)) $\vee$ *fst S* $\neq$ *None*›
shows ‹*no-step* (*next-model-filtered P*) *S*›
⟨*proof*⟩

end
theory *Watched-Literals-Transition-System-Enumeration*
imports *Watched-Literals.Watched-Literals-Transition-System Model-Enumeration*
begin

Design decision: we favour shorter clauses to (potentially) better models.

More precisely, we take the clause composed of decisions, instead of taking the full trail. This creates shorter clauses. However, this makes satisfying the initial clauses *harder* since fewer literals can be left undefined or be defined with the wrong sign.

For now there is no difference, since TWL produces only full models anyway. Remark that this is the clause that is produced by the minimization of the conflict of the full trail (except that this clauses would be learned and not added to the initial set of clauses, meaning that that the set of initial clauses is not harder to satisfy).

It is not clear if that would really make a huge performance difference.

The name DECO (e.g., *DECO-clause*) comes from Armin Biere's "decision only clauses" (DECO) optimisation (see Armin Biere's "Lingeling, Plingeling and Treengeling Entering the SAT Competition 2013"). If the learned clause becomes much larger that the clause normally learned by backjump, then the clause composed of the negation of the decision is learned instead (effectively doing a backtrack instead of a backjump). Unless we get more information from the filtering function, we are in the special case where the 1st-UIP is exactly the last decision.

An important property of the transition rules is that they violate the invariant that propagations are fully done before each decision. This means that we handle the transitions as a fast restart and not as a backjump as one would expect, since we cannot reuse any theorem about backjump.

definition *DECO-clause* :: ‹(′v, ′a) ann-lits ⇒ ′v clause›**where**
‹*DECO-clause M* = (*uminus o lit-of*) ‘# (*filter-mset is-decided* (*mset M*))›

lemma *distinct-mset-DECO*:
‹*distinct-mset* (*DECO-clause M*) ⟷ *distinct-mset* (*lit-of* ‘# *filter-mset is-decided* (*mset M*))›
(**is** ‹*?A* ⟷ *?B*›)
⟨*proof*⟩

lemma [*twl-st*]:
‹*init-clss* ($state_W$*-of T*) = *get-all-init-clss T*›
‹*learned-clss* ($state_W$*-of T*) = *get-all-learned-clss T*›
⟨*proof*⟩

lemma *atms-of-DECO-clauseD*:
‹*x* ∈ *atms-of* (*DECO-clause U*) ⟹ *x* ∈ *atms-of-s* (*lits-of-l U*)›
‹*x* ∈ *atms-of* (*DECO-clause U*) ⟹ *x* ∈ *atms-of* (*lit-of* ‘# *mset U*)›
⟨*proof*⟩

definition *TWL-DECO-clause* **where**
‹*TWL-DECO-clause M* =
TWL-Clause
((*uminus o lit-of*) ‘# *mset* (*take 2* (*filter is-decided M*)))
((*uminus o lit-of*) ‘# *mset* (*drop 2* (*filter is-decided M*)))›

lemma *clause-TWL-Deco-clause*[*simp*]: ‹*clause* (*TWL-DECO-clause M*) = *DECO-clause M*›
⟨*proof*⟩

inductive *negate-model-and-add-twl* :: ‹′v twl-st ⇒ ′v twl-st ⇒ bool› **where**
bj-unit:
‹*negate-model-and-add-twl* (*M*, *N*, *U*, *None*, *NP*, *UP*, *WS*, *Q*)
(*Propagated* (−*K*) (*DECO-clause M*) # *M1*, *N*, *U*, *None*, *add-mset* (*DECO-clause M*) *NP*, *UP*, {#}, {#*K*#})›
if
‹(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*get-level M K* = *count-decided M*› **and**
‹*count-decided M* = *1*› |
bj-nonunit:
‹*negate-model-and-add-twl* (*M*, *N*, *U*, *None*, *NP*, *UP*, *WS*, *Q*)
(*Propagated* (−*K*) (*DECO-clause M*) # *M1*, *add-mset* (*TWL-DECO-clause M*) *N*, *U*, *None*, *NP*, *UP*, {#},
{#*K*#})›
if
‹(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**
‹*get-level M K* = *count-decided M*› **and**
‹*count-decided M* ≥ *2*› |
restart-nonunit:
‹*negate-model-and-add-twl* (*M*, *N*, *U*, *None*, *NP*, *UP*, *WS*, *Q*)
(*M1*, *add-mset* (*TWL-DECO-clause M*) *N*, *U*, *None*, *NP*, *UP*, {#}, {#})›
if
‹(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition M*)› **and**

⟨*get-level M K < count-decided M*⟩ **and**
⟨*count-decided M > 1*⟩

Some remarks:

- Because of the invariants (unit clauses have to be propagated), a rule restart_unit would be the same as the bj_unit.
- The rules cleans the components about updates and do not assume that they are empty.

lemma *after-fast-restart-replay*:
assumes
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv (M′, N, U, None)*⟩ **and**
stgy-invs: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant (M′, N, U, None)*⟩ **and**
smaller-propa: ⟨$cdcl_W$*-restart-mset.no-smaller-propa (M′, N, U, None)*⟩ **and**
kept: ⟨*∀ L E. Propagated L E ∈ set (drop (length M′ − n) M′) ⟶ E ∈# N + U′*⟩ **and**
U′-U: ⟨*U′ ⊆# U*⟩ **and**
no-confl: ⟨*∀ C∈#N′. ∀ M1 K M2. M′ = M2 @ Decided K # M1 ⟶ ¬M1 ⊨as CNot C*⟩ **and**
no-propa: ⟨*∀ C∈#N′. ∀ M1 K M2 L. M′ = M2 @ Decided K # M1 ⟶ L ∈# C ⟶ ¬M1 ⊨as CNot (remove1-mset L C)*⟩
shows
⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy*** *([], N+N′, U′, None) (drop (length M′ − n) M′, N+ N′, U′, None)*⟩
⟨*proof*⟩

lemma *after-fast-restart-replay′*:
assumes
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv (M′, N, U, None)*⟩ **and**
stgy-invs: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy-invariant (M′, N, U, None)*⟩ **and**
smaller-propa: ⟨$cdcl_W$*-restart-mset.no-smaller-propa (M′, N, U, None)*⟩ **and**
kept: ⟨*∀ L E. Propagated L E ∈ set (drop (length M′ − n) M′) ⟶ E ∈# N + U′*⟩ **and**
U′-U: ⟨*U′ ⊆# U*⟩ **and**
N-N′: ⟨*N ⊆# N′*⟩ **and**
no-confl: ⟨*∀ C∈#N′−N. ∀ M1 K M2. M′ = M2 @ Decided K # M1 ⟶ ¬M1 ⊨as CNot C*⟩ **and**
no-propa: ⟨*∀ C∈#N′−N. ∀ M1 K M2 L. M′ = M2 @ Decided K # M1 ⟶ L ∈# C ⟶ ¬M1 ⊨as CNot (remove1-mset L C)*⟩
shows
⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-stgy*** *([], N′, U′, None) (drop (length M′ − n) M′, N′, U′, None)*⟩
⟨*proof*⟩

lemma *after-fast-restart-replay-no-stgy*:
assumes
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv (M′, N, U, None)*⟩ **and**
kept: ⟨*∀ L E. Propagated L E ∈ set (drop (length M′ − n) M′) ⟶ E ∈# N+N′ + U′*⟩ **and**
U′-U: ⟨*U′ ⊆# U*⟩
shows
⟨$cdcl_W$*-restart-mset.*$cdcl_W$** *([], N+N′, U′, None) (drop (length M′ − n) M′, N+N′, U′, None)*⟩
⟨*proof*⟩

lemma *after-fast-restart-replay-no-stgy′*:
assumes
inv: ⟨$cdcl_W$*-restart-mset.*$cdcl_W$*-all-struct-inv (M′, N, U, None)*⟩ **and**
kept: ⟨*∀ L E. Propagated L E ∈ set (drop (length M′ − n) M′) ⟶ E ∈# N′ + U′*⟩ **and**
U′-U: ⟨*U′ ⊆# U*⟩ **and**
⟨*N ⊆# N′*⟩
shows

‹$cdcl_W$-restart-mset.$cdcl_W$** ([], N′, U′, None) (drop (length M′ − n) M′, N′, U′, None)›
⟨proof⟩

lemma *$cdcl_W$-all-struct-inv-move-to-init*:
assumes *inv*: ‹$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N, U + U′, D)›
shows ‹$cdcl_W$-restart-mset.$cdcl_W$-all-struct-inv (M, N + U′, U, D)›
⟨proof⟩

lemma *twl-struct-invs-move-to-init*:
assumes ‹twl-struct-invs (M, N, U + U′, D, NP, UP, WS, Q)›
shows ‹twl-struct-invs (M, N + U′, U, D, NP, UP, WS, Q)›
⟨proof⟩

lemma *negate-model-and-add-twl-twl-struct-invs*:
fixes S T :: ‹′v twl-st›
assumes
‹negate-model-and-add-twl S T› **and**
‹twl-struct-invs S›
shows ‹twl-struct-invs T›
⟨proof⟩

lemma *get-all-ann-decomposition-count-decided-1*:
assumes
decomp: ‹(Decided K # M1, M2) ∈ set (get-all-ann-decomposition M)› **and**
count-dec: ‹count-decided M = 1›
shows ‹M = M2 @ Decided K # M1›
⟨proof⟩

lemma *negate-model-and-add-twl-twl-stgy-invs*:
assumes
‹negate-model-and-add-twl S T› **and**
‹twl-struct-invs S› **and**
‹twl-stgy-invs S›
shows ‹twl-stgy-invs T›
⟨proof⟩

lemma *cdcl-twl-stgy-$cdcl_W$-learned-clauses-entailed-by-init*:
assumes
‹cdcl-twl-stgy S s› **and**
‹twl-struct-invs S› **and**
‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of S)›
shows
‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of s)›
⟨proof⟩

lemma *rtranclp-cdcl-twl-stgy-$cdcl_W$-learned-clauses-entailed-by-init*:
assumes
‹cdcl-twl-stgy** S s› **and**
‹twl-struct-invs S› **and**
‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of S)›
shows
‹$cdcl_W$-restart-mset.$cdcl_W$-learned-clauses-entailed-by-init ($state_W$-of s)›
⟨proof⟩

lemma *negate-model-and-add-twl-$cdcl_W$-learned-clauses-entailed-by-init*:

assumes
‹*negate-model-and-add-twl S s*› **and**
‹*twl-struct-invs S*› **and**
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*)›
shows
‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of s*)›
⟨*proof*⟩

end
theory *Watched-Literals-Algorithm-Enumeration*
imports *Watched-Literals.Watched-Literals-Algorithm Watched-Literals-Transition-System-Enumeration*
begin

definition *cdcl-twl-enum-inv* :: ‹*'v twl-st* $\Rightarrow$ *bool*› **where**
‹*cdcl-twl-enum-inv S* $\longleftrightarrow$ *twl-struct-invs S* $\wedge$ *twl-stgy-invs S* $\wedge$ *final-twl-state S* $\wedge$
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*)›

definition *mod-restriction* :: ‹*'v clauses* $\Rightarrow$ *'v clauses* $\Rightarrow$ *bool*› **where**
‹*mod-restriction N N'* $\longleftrightarrow$
($\forall$ *M*. *M* $\models$*sm N* $\longrightarrow$ *M* $\models$*sm N'*) $\wedge$
($\forall$ *M*. *total-over-m M* (*set-mset N'*) $\longrightarrow$ *consistent-interp M* $\longrightarrow$ *M* $\models$*sm N'* $\longrightarrow$ *M* $\models$*sm N*)›

lemma *mod-restriction-satisfiable-iff*:
‹*mod-restriction N N'* $\Longrightarrow$ *satisfiable* (*set-mset N*) $\longleftrightarrow$ *satisfiable* (*set-mset N'*)›
⟨*proof*⟩

definition *enum-mod-restriction-st-clss* :: ‹(*'v twl-st* $\times$ (*'v literal list option* $\times$ *'v clauses*)) *set*› **where**
‹*enum-mod-restriction-st-clss* = {(*S*, (*M*, *N*)). *mod-restriction* (*get-all-init-clss S*) *N* $\wedge$
twl-struct-invs S $\wedge$ *twl-stgy-invs S* $\wedge$
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*) $\wedge$
atms-of-mm (*get-all-init-clss S*) = *atms-of-mm N*}›

definition *enum-model-st-direct* :: ‹(*'v twl-st* $\times$ (*'v literal list option* $\times$ *'v clauses*)) *set*› **where**
‹*enum-model-st-direct* = {(*S*, (*M*, *N*)).
mod-restriction (*get-all-init-clss S*) *N* $\wedge$
(*get-conflict S* = *None* $\longrightarrow$ *M* $\neq$ *None* $\wedge$ *lit-of* ‘# *mset* (*get-trail S*) = *mset* (*the M*)) $\wedge$
(*get-conflict S* $\neq$ *None* $\longrightarrow$ *M* = *None*) $\wedge$
atms-of-mm (*get-all-init-clss S*) = *atms-of-mm N* $\wedge$
(*get-conflict S* = *None* $\longrightarrow$ *next-model* (*map lit-of* (*get-trail S*)) *N*) $\wedge$
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*) $\wedge$
cdcl-twl-enum-inv S}›

definition *enum-model-st* :: ‹((*bool* $\times$ *'v twl-st*) $\times$ (*'v literal list option* $\times$ *'v clauses*)) *set*› **where**
‹*enum-model-st* = {((*b*, *S*), (*M*, *N*)).
mod-restriction (*get-all-init-clss S*) *N* $\wedge$
(*b* $\longrightarrow$ *get-conflict S* = *None* $\wedge$ *M* $\neq$ *None* $\wedge$ *lits-of-l* (*get-trail S*) = *set* (*the M*)) $\wedge$
(*get-conflict S* $\neq$ *None* $\longrightarrow$ $\neg$*b* $\wedge$ *M* = *None*)}›

fun *add-to-init-cls* :: ‹*'v twl-cls* $\Rightarrow$ *'v twl-st* $\Rightarrow$ *'v twl-st*› **where**
‹*add-to-init-cls C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) = (*M*, *add-mset C N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*)›

lemma *cdcl-twl-stgy-final-twl-stateE*:
assumes
‹$cdcl\text{-}twl\text{-}stgy^{**}$ *S T*› **and**

final: ‹*final-twl-state T*› **and**
‹*twl-struct-invs S*› **and**
‹*twl-stgy-invs S*› **and**
ent: ‹$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*)› **and**
Hunsat: ‹*get-conflict T* ≠ *None* ⟹ *unsatisfiable* (*set-mset* (*get-all-init-clss S*)) ⟹ *P*› **and**
Hsat: ‹*get-conflict T* = *None* ⟹ *consistent-interp* (*lits-of-l* (*get-trail T*)) ⟹
no-dup (*get-trail T*) ⟹ *atm-of* ‘ (*lits-of-l* (*get-trail T*)) ⊆ *atms-of-mm* (*get-all-init-clss T*) ⟹
get-trail T ⊨*asm get-all-init-clss S* ⟹ *satisfiable* (*set-mset* (*get-all-init-clss S*)) ⟹ *P*›
shows *P*
⟨*proof*⟩

context
fixes *P* :: ‹′*v literal set* ⇒ *bool*›
begin

fun *negate-model-and-add* :: ‹′*v literal list option* × ′*v clauses* ⇒ - × ′*v clauses*› **where**
‹*negate-model-and-add* (*Some M*, *N*) =
(*if P* (*set M*) *then* (*Some M*, *N*)
else (*None*, *add-mset* (*uminus* ‘# *mset M*) *N*))› |
‹*negate-model-and-add* (*None*, *N*) = (*None*, *N*)›

The code below is a little tricky to get right (in a way that can be easily refined later). There are three cases:

1. the considered clauses are not satisfiable. Then we can conclude that there is no model.
2. the considered clauses are satisfiable and there is at least one decision. Then, we can simply apply *negate-model-and-add-twl*.
3. the considered clauses are satisfiable and there are no decisions. Then we cannot apply *negate-model-and-add-twl*, because that would produce the empty clause that cannot be part of our state (because of our invariants). Therefore, as we know that the model is the last possible model, we break out of the loop and handle test if the model is acceptable outside of the loop.

definition *cdcl-twl-enum* :: ‹′*v twl-st* ⇒ *bool nres*› **where**
‹*cdcl-twl-enum S* = *do* {
S ← *conclusive-TWL-run S*;
S ← $WHILE_T{}^{cdcl\text{-}twl\text{-}enum\text{-}inv}$
(λ*S*. *get-conflict S* = *None* ∧ *count-decided*(*get-trail S*) > *0* ∧ ¬*P* (*lits-of-l* (*get-trail S*)))
(λ*S*. *do* {
S ← *SPEC* (*negate-model-and-add-twl S*);
conclusive-TWL-run S
})
S;
if get-conflict S = *None*
then RETURN (*if count-decided*(*get-trail S*) = *0 then P* (*lits-of-l* (*get-trail S*)) *else True*)
else RETURN (*False*)
}›

definition *next-model-filtered-nres* **where**
‹*next-model-filtered-nres N* =
SPEC (λ*b*. ∃ *M*. *full* (*next-model-filtered P*) *N M* ∧ *b* = (*fst M* ≠ *None*))›

lemma *mod-restriction-next-modelD*:
‹*mod-restriction N N′* ⟹ *atms-of-mm N* ⊆ *atms-of-mm N′* ⟹ *next-model M N* ⟹ *next-model M N′*›
⟨*proof*⟩

definition *enum-mod-restriction-st-clss-after* :: ‹(′*v twl-st* × (′*v literal list option* × ′*v clauses*)) *set*›
where
‹*enum-mod-restriction-st-clss-after* = {(*S*, (*M*, *N*)).
(*get-conflict S* = *None* ⟶ *count-decided* (*get-trail S*) = *0* ⟶
mod-restriction (*add-mset* {#} (*get-all-init-clss S*))
(*add-mset* (*uminus* ‘# *lit-of* ‘# *mset* (*get-trail S*)) *N*)) ∧
(*mod-restriction* (*get-all-init-clss S*) *N*) ∧
twl-struct-invs S ∧ *twl-stgy-invs S* ∧
(*get-conflict S* = *None* ⟶ *M* ≠ *None* ⟶ *P* (*set*(*the M*)) ∧ *lit-of* ‘# *mset* (*get-trail S*) = *mset* (*the M*)) ∧
(*get-conflict S* ≠ *None* ⟶ *M* = *None*) ∧
$cdcl_W$*-restart-mset.*$cdcl_W$*-learned-clauses-entailed-by-init* ($state_W$*-of S*) ∧
atms-of-mm (*get-all-init-clss S*) = *atms-of-mm N*}›

lemma *atms-of-uminus-lit-of*[*simp*]: ‹*atms-of* {#− *lit-of x*. *x* ∈# *A*#} = *atms-of* (*lit-of* ‘# *A*)›
⟨*proof*⟩

lemma *lit-of-mset-eq-mset-setD*[*dest*]:
‹*lit-of* ‘# *mset M* = *mset aa* ⟹ *set aa* = *lit-of* ‘ *set M*›
⟨*proof*⟩

lemma *mod-restriction-add-twice*[*simp*]:
‹*mod-restriction A* (*add-mset C* (*add-mset C N*)) ⟷ *mod-restriction A* (*add-mset C N*)›
⟨*proof*⟩

lemma
assumes
confl: ‹*get-conflict W* = *None*› **and**
count-dec: ‹*count-decided* (*get-trail W*) = *0*› **and**
enum-inv: ‹*cdcl-twl-enum-inv W*› **and**
mod-rest-U: ‹*mod-restriction* (*get-all-init-clss W*) *N*› **and**
atms-U-U′: ‹*atms-of-mm* (*get-all-init-clss W*) = *atms-of-mm N*›
shows
final-level0-add-empty-clause:
‹*mod-restriction* (*add-mset* {#} (*get-all-init-clss W*))
(*add-mset* {#− *lit-of x*. *x* ∈# *mset* (*get-trail W*)#} *N*)› (**is** *?A*) **and**
final-level0-add-empty-clause-unsat:
‹*unsatisfiable* (*set-mset* (*add-mset* {#− *lit-of x*. *x* ∈# *mset* (*get-trail W*)#} *N*))› (**is** *?B*)
⟨*proof*⟩

lemma *cdcl-twl-enum-next-model-filtered-nres*:
‹(*cdcl-twl-enum*, *next-model-filtered-nres*) ∈
[λ(*M*, *N*). *M* = *None*]$_f$ *enum-mod-restriction-st-clss* → ⟨*bool-rel*⟩*nres-rel*›
⟨*proof*⟩

end

end
theory *Watched-Literals-List-Enumeration*
imports *Watched-Literals-Algorithm-Enumeration Watched-Literals.Watched-Literals-List*

begin

lemma *convert-lits-l-filter-decided-uminus*: ‹$(S, S') \in$ *convert-lits-l M N* $\Longrightarrow$
map $(\lambda x. -$*lit-of x*$)$ (*filter is-decided S'*) = *map* $(\lambda x. -$*lit-of x*$)$ (*filter is-decided S*)›
⟨*proof*⟩

lemma *convert-lits-l-DECO-clause*[*simp*]:
‹$(S, S') \in$ *convert-lits-l M N* $\Longrightarrow$ *DECO-clause S'* = *DECO-clause S*›
⟨*proof*⟩

lemma *convert-lits-l-TWL-DECO-clause*[*simp*]:
‹$(S, S') \in$ *convert-lits-l M N* $\Longrightarrow$ *TWL-DECO-clause S'* = *TWL-DECO-clause S*›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹$(S, S') \in$ *twl-st-l b* $\Longrightarrow$ *DECO-clause* (*get-trail S'*) = *DECO-clause* (*get-trail-l S*)›
⟨*proof*⟩

lemma [*twl-st-l*]:
‹$(S, S') \in$ *twl-st-l b* $\Longrightarrow$ *TWL-DECO-clause* (*get-trail S'*) = *TWL-DECO-clause* (*get-trail-l S*)›
⟨*proof*⟩

lemma *DECO-clause-simp*[*simp*]:
‹*DECO-clause* (*A* @ *B*) = *DECO-clause A* + *DECO-clause B*›
‹*DECO-clause* (*Decided K* # *A*) = *add-mset* $(-K)$ (*DECO-clause A*)›
‹*DECO-clause* (*Propagated K C* # *A*) = *DECO-clause A*›
‹$(\bigwedge K.\ K \in$ *set A* $\Longrightarrow \neg$*is-decided K*$) \Longrightarrow$ *DECO-clause A* = {#}›
⟨*proof*⟩

definition *find-decomp-target* :: ‹*nat* ⇒ *'v twl-st-l* ⇒ (*'v twl-st-l* × *'v literal*) *nres*› **where**
‹*find-decomp-target* = (λ*i S*.
SPEC(λ(*T*, *K*). ∃ *M2 M1*. *equality-except-trail S T* ∧ *get-trail-l T* = *M1* ∧
(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-l S*)) ∧
get-level (*get-trail-l S*) *K* = *i*))›

fun *propagate-unit-and-add* :: ‹*'v literal* ⇒ *'v twl-st* ⇒ *'v twl-st*› **where**
‹*propagate-unit-and-add K* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) =
(*Propagated* $(-K)$ {#$-K$#} # *M*, *N*, *U*, *None*, *add-mset* {#$-K$#} *NE*, *UE*, {#}, {#*K*#})›

fun *propagate-unit-and-add-l* :: ‹*'v literal* ⇒ *'v twl-st-l* ⇒ *'v twl-st-l*› **where**
‹*propagate-unit-and-add-l K* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) =
(*Propagated* $(-K)$ *0* # *M*, *N*, *None*, *add-mset* {#$-K$#} *NE*, *UE*, {#}, {#*K*#})›

definition *negate-mode-bj-unit-l-inv* :: ‹*'v twl-st-l* ⇒ *bool*› **where**
‹*negate-mode-bj-unit-l-inv S* ⟷
(∃ (*S'*::*'v twl-st*) *b*. $(S, S') \in$ *twl-st-l b* ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'* ∧
twl-struct-invs S' ∧ *get-conflict-l S* = *None*)›

definition *negate-mode-bj-unit-l* :: ‹*'v twl-st-l* ⇒ *'v twl-st-l nres*› **where**
‹*negate-mode-bj-unit-l* = (λ*S*. *do* {
ASSERT(*negate-mode-bj-unit-l-inv S*);
(*S*, *K*) ← *find-decomp-target 1 S*;
RETURN (*propagate-unit-and-add-l K S*)
})›

lemma *negate-mode-bj-unit-l*:
fixes *S* :: ‹'*v twl-st-l*› **and** *S'* :: ‹'*v twl-st*›
assumes ‹*count-decided* (*get-trail-l S*) = *1*› **and**
SS': ‹(*S*, *S'*) ∈ *twl-st-l b*› **and**
struct-invs: ‹*twl-struct-invs S'*› **and**
add-inv: ‹*twl-list-invs S*› **and**
stgy-inv: ‹*twl-stgy-invs S'*› **and**
confl: ‹*get-conflict-l S* = *None*›
shows
‹*negate-mode-bj-unit-l S* ≤ ⇓{(*S*, *S''*). (*S*, *S''*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
clauses-to-update-l S = {#}}
(*SPEC* (*negate-model-and-add-twl S'*))›
⟨*proof*⟩

definition *DECO-clause-l* :: ‹('*v*, '*a*) *ann-lits* ⇒ '*v clause-l*› **where**
‹*DECO-clause-l M* = *map* (*uminus o lit-of*) (*filter is-decided M*)›

fun *propagate-nonunit-and-add* :: ‹'*v literal* ⇒ '*v literal multiset twl-clause* ⇒ '*v twl-st* ⇒ '*v twl-st*›
where
‹*propagate-nonunit-and-add K C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) = *do* {
(*Propagated* (−*K*) (*clause C*) # *M*, *add-mset C N*, *U*, *None*,
NE, *UE*, {#}, {#*K*#})
}›

fun *propagate-nonunit-and-add-l* :: ‹'*v literal* ⇒ '*v clause-l* ⇒ *nat* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l*› **where**
‹*propagate-nonunit-and-add-l K C i* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *do* {
(*Propagated* (−*K*) *i* # *M*, *fmupd i* (*C*, *True*) *N*, *None*,
NE, *UE*, {#}, {#*K*#})
}›

definition *negate-mode-bj-nonunit-l-inv* **where**
‹*negate-mode-bj-nonunit-l-inv S* ⟷
(∃ *S''* *b*. (*S*, *S''*) ∈ *twl-st-l b* ∧ *twl-list-invs S* ∧ *count-decided* (*get-trail-l S*) > *1* ∧
twl-struct-invs S'' ∧ *twl-stgy-invs S''* ∧ *get-conflict-l S* = *None*)›

definition *negate-mode-bj-nonunit-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*negate-mode-bj-nonunit-l* = (λ*S*. *do* {
ASSERT(*negate-mode-bj-nonunit-l-inv S*);
let C = *DECO-clause-l* (*get-trail-l S*);
(*S*, *K*) ← *find-decomp-target* (*count-decided* (*get-trail-l S*)) *S*;
i ← *get-fresh-index* (*get-clauses-l S*);
RETURN (*propagate-nonunit-and-add-l K C i S*)
})›

lemma *DECO-clause-l-DECO-clause*[*simp*]:
‹*mset* (*DECO-clause-l M1*) = *DECO-clause M1*›
⟨*proof*⟩

lemma *TWL-DECO-clause-alt-def*:
‹*TWL-DECO-clause M1* =
TWL-Clause (*mset* (*watched-l* (*DECO-clause-l M1*)))
(*mset* (*unwatched-l* (*DECO-clause-l M1*)))›
⟨*proof*⟩

lemma *length-DECO-clause-l*[*simp*]:
  ‹*length* (*DECO-clause-l M*) = *count-decided M*›
  ⟨*proof*⟩

lemma *negate-mode-bj-nonunit-l*:
  fixes *S* :: ‹'*v twl-st-l*› **and** *S'* :: ‹'*v twl-st*›
  assumes
    count-dec: ‹*count-decided* (*get-trail-l S*) > *1*› **and**
    SS': ‹(*S*, *S'*) ∈ *twl-st-l b*› **and**
    struct-invs: ‹*twl-struct-invs S'*› **and**
    add-inv: ‹*twl-list-invs S*› **and**
    stgy-inv: ‹*twl-stgy-invs S'*› **and**
    confl: ‹*get-conflict-l S* = *None*›
  shows
    ‹*negate-mode-bj-nonunit-l S* ≤ ⇓{(*S*, *S''*). (*S*, *S''*) ∈ *twl-st-l None* ∧ *twl-list-invs S* ∧
        clauses-to-update-l S = {#}}
      (*SPEC* (*negate-model-and-add-twl S'*))›
⟨*proof*⟩

fun *restart-nonunit-and-add* :: ‹'*v literal multiset twl-clause* ⇒ '*v twl-st* ⇒ '*v twl-st*› **where**
  ‹*restart-nonunit-and-add C* (*M*, *N*, *U*, *D*, *NE*, *UE*, *WS*, *Q*) = *do* {
      (*M*, *add-mset C N*, *U*, *None*, *NE*, *UE*, {#}, {#})
    }›

fun *restart-nonunit-and-add-l* :: ‹'*v clause-l* ⇒ *nat* ⇒ '*v twl-st-l* ⇒ '*v twl-st-l*› **where**
  ‹*restart-nonunit-and-add-l C i* (*M*, *N*, *D*, *NE*, *UE*, *WS*, *Q*) = *do* {
      (*M*, *fmupd i* (*C*, *True*) *N*, *None*, *NE*, *UE*, {#}, {#})
    }›

definition *negate-mode-restart-nonunit-l-inv* :: ‹'*v twl-st-l* ⇒ *bool*› **where**
‹*negate-mode-restart-nonunit-l-inv S* ⟷
  (∃ *S'* *b*. (*S*, *S'*) ∈ *twl-st-l b* ∧ *twl-struct-invs S'* ∧ *twl-list-invs S* ∧ *twl-stgy-invs S'* ∧
    count-decided (*get-trail-l S*) > *1* ∧ *get-conflict-l S* = *None*)›

definition *negate-mode-restart-nonunit-l* :: ‹'*v twl-st-l* ⇒ '*v twl-st-l nres*› **where**
‹*negate-mode-restart-nonunit-l* = (λ*S*. *do* {
    ASSERT(*negate-mode-restart-nonunit-l-inv S*);
    let C = *DECO-clause-l* (*get-trail-l S*);
    i ← *SPEC*(λ*i*. *i* < *count-decided* (*get-trail-l S*));
    (*S*, *K*) ← *find-decomp-target i S*;
    i ← *get-fresh-index* (*get-clauses-l S*);
    RETURN (*restart-nonunit-and-add-l C i S*)
  })›

lemma *negate-mode-restart-nonunit-l*:
  fixes *S* :: ‹'*v twl-st-l*› **and** *S'* :: ‹'*v twl-st*›
  assumes
    count-dec: ‹*count-decided* (*get-trail-l S*) > *1*› **and**
    SS': ‹(*S*, *S'*) ∈ *twl-st-l b*› **and**
    struct-invs: ‹*twl-struct-invs S'*› **and**
    add-inv: ‹*twl-list-invs S*› **and**
    stgy-inv: ‹*twl-stgy-invs S'*› **and**
    confl: ‹*get-conflict-l S* = *None*›
  shows

‹negate-mode-restart-nonunit-l S ≤ ⇓{(S, S''). (S, S'') ∈ twl-st-l None ∧ twl-list-invs S ∧
    clauses-to-update-l S = {#}}
   (SPEC (negate-model-and-add-twl S'))›
⟨proof⟩

definition *negate-mode-l-inv* **where**
‹negate-mode-l-inv S ⟷
  (∃ S' b. (S, S') ∈ twl-st-l b ∧ twl-struct-invs S' ∧ twl-list-invs S ∧ twl-stgy-invs S' ∧
   get-conflict-l S = None ∧ count-decided (get-trail-l S) ≠ 0)›

definition *negate-mode-l* :: ‹'v twl-st-l ⇒ 'v twl-st-l nres› **where**
‹negate-mode-l S = do {
  ASSERT(negate-mode-l-inv S);
  if count-decided (get-trail-l S) = 1
  then negate-mode-bj-unit-l S
  else do {
    b ← SPEC(λ-. True);
    if b then negate-mode-bj-nonunit-l S else negate-mode-restart-nonunit-l S
  }
}›

lemma *negate-mode-l*:
fixes S :: ‹'v twl-st-l› **and** S' :: ‹'v twl-st›
assumes
  SS': ‹(S, S') ∈ twl-st-l b› **and**
  struct-invs: ‹twl-struct-invs S'› **and**
  add-inv: ‹twl-list-invs S› **and**
  stgy-inv: ‹twl-stgy-invs S'› **and**
  confl: ‹get-conflict-l S = None› **and**
  ‹count-decided (get-trail-l S) ≠ 0›
shows
  ‹negate-mode-l S ≤ ⇓{(S, S''). (S, S'') ∈ twl-st-l None ∧ twl-list-invs S ∧
    clauses-to-update-l S = {#}}
   (SPEC (negate-model-and-add-twl S'))›
⟨proof⟩

context
fixes P :: ‹'v literal set ⇒ bool›
begin

definition *cdcl-twl-enum-inv-l* :: ‹'v twl-st-l ⇒ bool› **where**
‹cdcl-twl-enum-inv-l S ⟷
  (∃ S'. (S, S') ∈ twl-st-l None ∧ cdcl-twl-enum-inv S') ∧
   twl-list-invs S›

definition *cdcl-twl-enum-l* :: ‹'v twl-st-l ⇒ bool nres› **where**
‹cdcl-twl-enum-l S = do {
   S ← cdcl-twl-stgy-prog-l S;
   S ← $WHILE_T^{cdcl\text{-}twl\text{-}enum\text{-}inv\text{-}l}$
    (λS. get-conflict-l S = None ∧ count-decided(get-trail-l S) > 0 ∧
       ¬P (lits-of-l (get-trail-l S)))
    (λS. do {
        S ← negate-mode-l S;
        cdcl-twl-stgy-prog-l S
      })
    S;

if get-conflict-l S = None
then RETURN (if count-decided(get-trail-l S) = 0 then P (lits-of-l (get-trail-l S)) else True)
else RETURN (False)
})

lemma *negate-model-and-add-twl-resultD*:
⟨*negate-model-and-add-twl S T* ⟹
clauses-to-update T = {#} ∧ *get-conflict T* = *None*⟩
⟨*proof*⟩

lemma *cdcl-twl-enum-l*:
fixes *S* :: ⟨'*v twl-st-l*⟩ **and** *S'* :: ⟨'*v twl-st*⟩
assumes
SS': ⟨(*S*, *S'*) ∈ *twl-st-l None*⟩ **and**
struct-invs: ⟨*twl-struct-invs S'*⟩ **and**
add-inv: ⟨*twl-list-invs S*⟩ **and**
stgy-inv: ⟨*twl-stgy-invs S'*⟩ **and**
confl: ⟨*get-conflict-l S* = *None*⟩ **and**
⟨*count-decided* (*get-trail-l S*) ≠ *0*⟩ **and**
⟨*clauses-to-update-l S* = {#}⟩
shows
⟨*cdcl-twl-enum-l S* ≤ ⇓ *bool-rel*
(*cdcl-twl-enum P S'*)⟩
⟨*proof*⟩

end

end
theory *Watched-Literals-Watch-List-Enumeration*
imports *Watched-Literals-List-Enumeration Watched-Literals.Watched-Literals-Watch-List*
begin

definition *find-decomp-target-wl* :: ⟨*nat* ⇒ '*v twl-st-wl* ⇒ ('*v twl-st-wl* × '*v literal*) *nres*⟩ **where**
⟨*find-decomp-target-wl* = (λ*i S*.
SPEC(λ(*T*, *K*). ∃ *M2 M1*. *equality-except-trail-wl S T* ∧ *get-trail-wl T* = *M1* ∧
(*Decided K* # *M1*, *M2*) ∈ *set* (*get-all-ann-decomposition* (*get-trail-wl S*)) ∧
get-level (*get-trail-wl S*) *K* = *i*))⟩

fun *propagate-unit-and-add-wl* :: ⟨'*v literal* ⇒ '*v twl-st-wl* ⇒ '*v twl-st-wl*⟩ **where**
⟨*propagate-unit-and-add-wl K* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) =
(*Propagated* (−*K*) *0* # *M*, *N*, *None*, *add-mset* {#−*K*#} *NE*, *UE*, {#*K*#}, *W*)⟩

definition *negate-mode-bj-unit-wl* :: ⟨'*v twl-st-wl* ⇒ '*v twl-st-wl nres*⟩ **where**
⟨*negate-mode-bj-unit-wl* = (λ*S*. *do* {
(*S*, *K*) ← *find-decomp-target-wl 1 S*;
ASSERT(*K* ∈# *all-lits-of-mm* (*clause* '# *twl-clause-of* '# *ran-mf* (*get-clauses-wl S*) +
get-unit-clauses-wl S));
RETURN (*propagate-unit-and-add-wl K S*)
})⟩

abbreviation *find-decomp-target-wl-ref* **where**
⟨*find-decomp-target-wl-ref S* ≡
{((*T*, *K*), (*T'*, *K'*)). (*T*, *T'*) ∈ {(*T*, *T'*). (*T*, *T'*) ∈ *state-wl-l None* ∧ *correct-watching T*} ∧
(*K* , *K'*) ∈ *Id* ∧
K ∈# *all-lits-of-mm* (*clause* '# *twl-clause-of* '# *ran-mf* (*get-clauses-wl T*) +

get-unit-clauses-wl T) ∧
K ∈# *all-lits-of-mm* (*clause* ‘# *twl-clause-of* ‘# *ran-mf* (*get-clauses-wl T*) +
get-unit-init-clss-wl T) ∧ *equality-except-trail-wl S T* ∧
atms-of (*DECO-clause* (*get-trail-wl S*)) ⊆ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *ran-mf* (*get-clauses-wl T*) +
get-unit-init-clss-wl T) ∧ *distinct-mset* (*DECO-clause* (*get-trail-wl S*)) ∧
correct-watching T}›

lemma *DECO-clause-nil*[*simp*]: ‹*DECO-clause* [] = {#}›
⟨*proof*⟩

lemma *in-DECO-clauseD*: ‹*x* ∈# *DECO-clause M* ⟹ −*x* ∈ *lits-of-l M*›
⟨*proof*⟩

lemma *in-atms-of-DECO-clauseD*: ‹*x* ∈ *atms-of* (*DECO-clause M*) ⟹ *x* ∈ *atm-of* ‘ (*lits-of-l M*)›
⟨*proof*⟩

lemma *no-dup-distinct-mset-DECO-clause*:
assumes ‹*no-dup M*›
shows ‹*distinct-mset* (*DECO-clause M*)›
⟨*proof*⟩

lemma *find-decomp-target-wl-find-decomp-target-l*:
assumes
SS′: ‹(*S*, *S′*) ∈ {(*S*, *S′′*). (*S*, *S′′*) ∈ *state-wl-l None* ∧ *correct-watching S*}› **and**
inv: ‹∃ *S′′ b*. (*S′*, *S′′*) ∈ *twl-st-l b* ∧ *twl-struct-invs S′′*› **and**
[*simp*]: ‹*a* = *a′*›
shows ‹*find-decomp-target-wl a S* ≤
⇓ (*find-decomp-target-wl-ref S*) (*find-decomp-target a′ S′*)›
(**is** ‹- ≤ ⇓ *?negate* -›)
⟨*proof*⟩

lemma *negate-mode-bj-unit-wl-negate-mode-bj-unit-l*:
fixes *S* :: ‹′*v twl-st-wl*› **and** *S′* :: ‹′*v twl-st-l*›
assumes ‹*count-decided* (*get-trail-wl S*) = *1*› **and**
SS′: ‹(*S*, *S′*) ∈ {(*S*, *S′*). (*S*, *S′*) ∈ *state-wl-l None* ∧ *correct-watching S*}›
shows
‹*negate-mode-bj-unit-wl S* ≤ ⇓{(*S*, *S′*). (*S*, *S′*) ∈ *state-wl-l None* ∧ *correct-watching S*}
(*negate-mode-bj-unit-l S′*)›
(**is** ‹- ≤ ⇓ *?R* -›)
⟨*proof*⟩

definition *propagate-nonunit-and-add-wl-pre*
:: ‹′*v literal* ⇒ ′*v clause-l* ⇒ *nat* ⇒ ′*v twl-st-wl* ⇒ *bool*› **where**
‹*propagate-nonunit-and-add-wl-pre K C i S* ⟷
length C ≥ *2* ∧ *i* > *0* ∧ *i* ∉# *dom-m* (*get-clauses-wl S*) ∧
atms-of (*mset C*) ⊆ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *ran-mf* (*get-clauses-wl S*) +
get-unit-init-clss-wl S)›

fun *propagate-nonunit-and-add-wl*
:: ‹′*v literal* ⇒ ′*v clause-l* ⇒ *nat* ⇒ ′*v twl-st-wl* ⇒ ′*v twl-st-wl nres*›
where
‹*propagate-nonunit-and-add-wl K C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *do* {
ASSERT(*propagate-nonunit-and-add-wl-pre K C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*));
let b = (*length C* = *2*);
let W = *W*(*C*!*0* := *W* (*C*!*0*) @ [(*i*, *C*!*1*, *b*)]);

let W = W(C!1 := W (C!1) @ [(i, C!0, b)]);
RETURN (Propagated (−K) i # M, fmupd i (C, True) N, None,
NE, UE, {#K#}, W)
})›

lemma *twl-st-l-splitD*:
‹(⋀*M N D NE UE Q W. f (M, N, D, NE, UE, Q, W) = P M N D NE UE Q W*) ⟹
f S = P (get-trail-l S) (get-clauses-l S) (get-conflict-l S) (get-unit-init-clauses-l S)
(get-unit-learned-clauses-l S) (clauses-to-update-l S) (literals-to-update-l S)›
⟨*proof*⟩

lemma *twl-st-wl-splitD*:
‹(⋀*M N D NE UE Q W. f (M, N, D, NE, UE, Q, W) = P M N D NE UE Q W*) ⟹
f S = P (get-trail-wl S) (get-clauses-wl S) (get-conflict-wl S) (get-unit-init-clss-wl S)
(get-unit-learned-clss-wl S) (literals-to-update-wl S) (get-watched-wl S)›
⟨*proof*⟩

definition *negate-mode-bj-nonunit-wl-inv* **where**
‹*negate-mode-bj-nonunit-wl-inv S* ⟷
(∃ *S'' b*. (*S*, *S''*) ∈ *state-wl-l b* ∧ *negate-mode-bj-nonunit-l-inv S''* ∧ *correct-watching S*)›

definition *negate-mode-bj-nonunit-wl* :: ‹*'v twl-st-wl ⇒ 'v twl-st-wl nres*› **where**
‹*negate-mode-bj-nonunit-wl* = (λ*S. do* {
ASSERT(negate-mode-bj-nonunit-wl-inv S);
let C = DECO-clause-l (get-trail-wl S);
(S, K) ← find-decomp-target-wl (count-decided (get-trail-wl S)) S;
i ← get-fresh-index-wl (get-clauses-wl S) (get-unit-clauses-wl S) (get-watched-wl S);
propagate-nonunit-and-add-wl K C i S
})›

lemmas *propagate-nonunit-and-add-wl-def* =
twl-st-wl-splitD[of ‹*propagate-nonunit-and-add-wl - - -*›, *OF propagate-nonunit-and-add-wl.simps*]

lemmas *propagate-nonunit-and-add-l-def* =
twl-st-l-splitD[of ‹*propagate-nonunit-and-add-l - - -*›, *OF propagate-nonunit-and-add-l.simps*,
rule-format]

lemma *atms-of-subset-in-atms-ofI*:
‹*atms-of C* ⊆ *atms-of-ms N* ⟹ *L* ∈# *C* ⟹ *atm-of L* ∈ *atms-of-ms N*›
⟨*proof*⟩

lemma *in-DECO-clause-l-in-DECO-clause-iff*:
‹*x* ∈ *set (DECO-clause-l M)* ⟷ *x* ∈# *(DECO-clause M)*›
⟨*proof*⟩

lemma *distinct-DECO-clause-l*:
‹*no-dup M* ⟹ *distinct (DECO-clause-l M)*›
⟨*proof*⟩

lemma *propagate-nonunit-and-add-wl-propagate-nonunit-and-add-l*:
assumes
SS': ‹(*S*, *S'*) ∈ *state-wl-l None*› **and**
inv: ‹*negate-mode-bj-nonunit-wl-inv S*› **and**
TK: ‹(*TK*, *TK'*) ∈ *find-decomp-target-wl-ref S*› **and**
[*simp*]: ‹*TK'* = (*T*, *K*)› **and**

[*simp*]: ‹*TK* = (*T′*, *K′*)› **and**
ij: ‹(*i*, *j*) ∈ {(*i*, *j*). *i* = *j* ∧ *i* ∉# *dom-m* (*get-clauses-wl T′*) ∧ *i* > *0* ∧
(∀ *L* ∈# *all-lits-of-mm* (*mset* ‘# *ran-mf* (*get-clauses-wl T′*) + *get-unit-clauses-wl T′*) .
i ∉ *fst* ‘ *set* (*watched-by T′ L*))}›
shows ‹*propagate-nonunit-and-add-wl K′* (*DECO-clause-l* (*get-trail-wl S*)) *i T′*
≤ *SPEC* (λ*c*. (*c*, *propagate-nonunit-and-add-l K*
(*DECO-clause-l* (*get-trail-l S′*)) *j T*)
∈ {(*S*, *S″*).
(*S*, *S″*) ∈ *state-wl-l None* ∧ *correct-watching S*})›
⟨*proof*⟩

lemma *watched-by-alt-def*:
‹*watched-by T L* = *get-watched-wl T L*›
⟨*proof*⟩

lemma *negate-mode-bj-nonunit-wl-negate-mode-bj-nonunit-l*:
fixes *S* :: ‹*′v twl-st-wl*› **and** *S′* :: ‹*′v twl-st-l*›
assumes
SS′: ‹(*S*, *S′*) ∈ {(*S*, *S″*). (*S*, *S″*) ∈ *state-wl-l None* ∧ *correct-watching S*}›
shows
‹*negate-mode-bj-nonunit-wl S* ≤ ⇓{(*S*, *S″*). (*S*, *S″*) ∈ *state-wl-l None* ∧ *correct-watching S*}
(*negate-mode-bj-nonunit-l S′*)›
⟨*proof*⟩

definition *negate-mode-restart-nonunit-wl-inv* :: ‹*′v twl-st-wl* ⇒ *bool*› **where**
‹*negate-mode-restart-nonunit-wl-inv S* ⟷
(∃ *S′ b*. (*S*, *S′*) ∈ *state-wl-l b* ∧ *negate-mode-restart-nonunit-l-inv S′* ∧ *correct-watching S*)›

definition *restart-nonunit-and-add-wl-inv* **where**
‹*restart-nonunit-and-add-wl-inv C i S* ⟷
length C ≥ *2* ∧ *correct-watching S* ∧
atms-of (*mset C*) ⊆ *atms-of-mm* (*clause* ‘# *twl-clause-of* ‘# *ran-mf* (*get-clauses-wl S*) +
get-unit-init-clss-wl S)›

fun *restart-nonunit-and-add-wl* :: ‹*′v clause-l* ⇒ *nat* ⇒ *′v twl-st-wl* ⇒ *′v twl-st-wl nres*› **where**
‹*restart-nonunit-and-add-wl C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*) = *do* {
ASSERT(*restart-nonunit-and-add-wl-inv C i* (*M*, *N*, *D*, *NE*, *UE*, *Q*, *W*));
let b = (*length C* = *2*);
let W = *W*(*C*!*0* := *W* (*C*!*0*) @ [(*i*, *C*!*1*, *b*)]);
let W = *W*(*C*!*1* := *W* (*C*!*1*) @ [(*i*, *C*!*0*, *b*)]);
RETURN (*M*, *fmupd i* (*C*, *True*) *N*, *None*, *NE*, *UE*, {#}, *W*)
}›

definition *negate-mode-restart-nonunit-wl* :: ‹*′v twl-st-wl* ⇒ *′v twl-st-wl nres*› **where**
‹*negate-mode-restart-nonunit-wl* = (λ*S*. *do* {
ASSERT(*negate-mode-restart-nonunit-wl-inv S*);
let C = *DECO-clause-l* (*get-trail-wl S*);
i ← *SPEC*(λ*i*. *i* < *count-decided* (*get-trail-wl S*));
(*S*, *K*) ← *find-decomp-target-wl i S*;
i ← *get-fresh-index-wl* (*get-clauses-wl S*) (*get-unit-clauses-wl S*) (*get-watched-wl S*);
restart-nonunit-and-add-wl C i S
})›

definition *negate-mode-wl-inv* **where**
‹*negate-mode-wl-inv S* ⟷

(∃ S′ b. (S, S′) ∈ state-wl-l b ∧ negate-mode-l-inv S′ ∧ correct-watching S)›

definition *negate-mode-wl* :: ‹′v twl-st-wl ⇒ ′v twl-st-wl nres› **where**
‹negate-mode-wl S = do {
  ASSERT(negate-mode-wl-inv S);
  if count-decided (get-trail-wl S) = 1
  then negate-mode-bj-unit-wl S
  else do {
    b ← SPEC(λ-. True);
    if b then negate-mode-bj-nonunit-wl S else negate-mode-restart-nonunit-wl S
  }
})›

lemma *correct-watching-learn-no-propa*:
assumes
L1: ‹atm-of L1 ∈ atms-of-mm (mset ‘# ran-mf N + NE)› **and**
L2: ‹atm-of L2 ∈ atms-of-mm (mset ‘# ran-mf N + NE)› **and**
UW: ‹atms-of (mset UW) ⊆ atms-of-mm (mset ‘# ran-mf N + NE)› **and**
‹L1 ≠ L2› **and**
i-dom: ‹i ∉# dom-m N› **and**
‹⋀L. L ∈# all-lits-of-mm (mset ‘# ran-mf N + (NE + UE)) ⟹ i ∉ fst ‘ set (W L)› **and**
‹b ⟷ length (L1 # L2 # UW) = 2›
shows
‹correct-watching (M, fmupd i (L1 # L2 # UW, b′) N,
  D, NE, UE, Q, W (L1 := W L1 @ [(i, L2, b)], L2 := W L2 @ [(i, L1, b)])) ⟷
correct-watching (M, N, D, NE, UE, Q, W)›
⟨proof⟩

lemma *restart-nonunit-and-add-wl-restart-nonunit-and-add-l*:
assumes
SS′: ‹(S, S′) ∈ {(S, S′). (S, S′) ∈ state-wl-l None ∧ correct-watching S}› **and**
l-inv: ‹negate-mode-restart-nonunit-l-inv S′› **and**
inv: ‹negate-mode-restart-nonunit-wl-inv S› **and**
‹(m, n) ∈ nat-rel› **and**
‹m ∈ {i. i < count-decided (get-trail-wl S)}› **and**
‹n ∈ {i. i < count-decided (get-trail-l S′)}› **and**
TK: ‹(TK, TK′) ∈ find-decomp-target-wl-ref S› **and**
[simp]: ‹TK′ = (T, K)› **and**
[simp]: ‹TK = (T′, K′)› **and**
ij: ‹(i, j) ∈ {(i, j). i = j ∧ i ∉# dom-m (get-clauses-wl T′) ∧ i > 0 ∧
  (∀ L ∈# all-lits-of-mm (mset ‘# ran-mf (get-clauses-wl T′) + get-unit-clauses-wl T′) .
    i ∉ fst ‘ set (watched-by T′ L))}›
shows ‹restart-nonunit-and-add-wl (DECO-clause-l (get-trail-wl S)) i T′
  ≤ SPEC (λc. (c, restart-nonunit-and-add-l
      (DECO-clause-l (get-trail-l S′)) j T)
    ∈ {(S, S′′).
      (S, S′′) ∈ state-wl-l None ∧ correct-watching S})›
⟨proof⟩

lemma *negate-mode-restart-nonunit-wl-negate-mode-restart-nonunit-l*:
fixes S :: ‹′v twl-st-wl› **and** S′ :: ‹′v twl-st-l›
assumes
SS′: ‹(S, S′) ∈ {(S, S′′). (S, S′′) ∈ state-wl-l None ∧ correct-watching S}›
shows
‹negate-mode-restart-nonunit-wl S ≤
  ⇓ {(S, S′′). (S, S′′) ∈ state-wl-l None ∧ correct-watching S}

(*negate-mode-restart-nonunit-l* S')›
⟨*proof*⟩

lemma *negate-mode-wl-negate-mode-l*:
fixes S :: ‹'*v twl-st-wl*› **and** S' :: ‹'*v twl-st-l*›
assumes
SS': ‹$(S, S') \in \{(S, S'')$. $(S, S'') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\}$› **and**
confl: ‹*get-conflict-wl* S = *None*›
shows
‹*negate-mode-wl* $S \leq$
$\Downarrow$ $\{(S, S'')$. $(S, S'') \in$ *state-wl-l None* $\wedge$ *correct-watching* $S\}$
(*negate-mode-l* S')›
⟨*proof*⟩

context
fixes P :: ‹'*v literal set* $\Rightarrow$ *bool*›
begin

definition *cdcl-twl-enum-inv-wl* :: ‹'*v twl-st-wl* $\Rightarrow$ *bool*› **where**
‹*cdcl-twl-enum-inv-wl* $S \longleftrightarrow$
($\exists S'$. $(S, S') \in$ *state-wl-l None* $\wedge$ *cdcl-twl-enum-inv-l* S') $\wedge$
correct-watching S›

definition *cdcl-twl-enum-wl* :: ‹'*v twl-st-wl* $\Rightarrow$ *bool nres*› **where**
‹*cdcl-twl-enum-wl* S = *do* {
$S \leftarrow$ *cdcl-twl-stgy-prog-wl* S;
$S \leftarrow$ $WHILE_T{}^{cdcl\text{-}twl\text{-}enum\text{-}inv\text{-}wl}$
(λS. *get-conflict-wl* S = *None* $\wedge$ *count-decided*(*get-trail-wl* S) > *0* $\wedge$
$\neg P$ (*lits-of-l* (*get-trail-wl* S)))
(λS. *do* {
$S \leftarrow$ *negate-mode-wl* S;
cdcl-twl-stgy-prog-wl S
})
S;
if get-conflict-wl S = *None*
then RETURN (*if count-decided*(*get-trail-wl* S) = *0 then* P (*lits-of-l* (*get-trail-wl* S)) *else True*)
else RETURN (*False*)
}›

lemma *cdcl-twl-enum-wl-cdcl-twl-enum-l*:
assumes
SS': ‹$(S, S') \in$ *state-wl-l None*› **and**
corr: ‹*correct-watching* S›
shows
‹*cdcl-twl-enum-wl* $S \leq \Downarrow$ *bool-rel*
(*cdcl-twl-enum-l* P S')›
⟨*proof*⟩

end

end