***ISRN GEONUM-NST--2001-03--FR*, 2001.**
F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2.
GEOTOP, Université du Québec à Montréal, 1997.

Spherical Discrete Element Code
SDEC V. 2.00

Manuel d'utilisation

(n'incluant pas la référence «en ligne » livrée avec le logiciel »)

F.V. Donzé

***ISRN GEONUM-NST--2001-03--FR*, 2001.**
F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2.
GEOTOP, Université du Québec à Montréal, 1997.

PARTIE 1

Méthode des Eléments Discrets : formulation utilisée dans SDEC

Sommaire

Introduction

Le présent code *SDEC V 2.00* [1] utilise une modélisation numérique fondée sur la *Méthode des Eléments Discrets* (*MED*). Cette méthode permet de caractériser la localisation de la déformation et de suivre la création et l'évolution des fractures lors de fortes sollicitations dynamiques jusqu'à l'éventuelle fragmentation du milieu (Donzé et *al*., 1996).

Principe de la Méthode des Eléments Discrets

Deux approches existent pour simuler la déformation dans les géomatériaux : l'approche continue, où l'on s'intéresse au comportement du milieu dans son ensemble et l'approche discrète où le milieu est considéré comme discontinu et représenté par l'assemblage d'un ensemble d'éléments modélisé (Müller, 1996).

L'approche continue ne s'applique pas à tous les cas d'étude : il y est en effet difficile d'analyser des phénomènes comme la ségrégation, la fracturation ou encore la fragmentation. C'est pourquoi, au vu de l'application présente, nous avons choisi l'approche discrète. Les trois étapes principales de cette approche consistent :

- D'abord définir la géométrie du milieu, à en donner ses caractéristiques.
- Dans un deuxième temps, l'algorithme doit détecter et définir les liens entre ces éléments. Le nombre d'opérations nécessaires pour ce faire dépendra du nombre d'éléments et de leurs formes ainsi que de leur répartition spatiale.
- Enfin, on donnera les lois physiques auxquelles obéissent les éléments, le type de forces d'interaction le comportement des éléments à l'état initiale puis lors d'un contact avec un autre élément.

Si ces lois physiques sont celles de la mécanique classique, on parlera d'une approche discrète newtonienne. Deux grandes écoles dominent cette dernière approche: celle dite des corps déformables et celle des corps indéformables (Müller, 1996). Dans la première, lorsque deux éléments sont en contact, ils peuvent « se chevaucher ». Les contacts ont une durée non nulle et les interactions entre les éléments varient de façon continue au cours du chevauchement. Pour calculer l'évolution d'un tel système, il d'agira d'intégrer des équations différentielles du second ordre.

Dans l'école des corps indéformables, le choc est considéré comme instantané. Le contact a lieu en un point et les éléments ne peuvent pas se chevaucher.

La méthode utilisée dans *SDEC V 2.0*, s'apparente à l'approche discrète newtonienne utilisant des corps déformables. Elle est généralement appelée la méthode des éléments discrets. D'autres auteurs (Cundall, 1971; Cundall & Strack 1979; Cundall, 1988; Hart et al. 1988, Walton, 1993), utilisent une approche sensiblement similaire (méthode des éléments distincts) avec cependant des variantes importantes que nous citerons au fur et à mesure du développement de l'algorithme.

Dans la méthode des éléments discrets de *SDEC V 2.0*, les éléments sont sphériques et sont de volume fini. Deux éléments peuvent être en interaction sans être en contact. Tout élément se trouvant à l'intérieur d'un rayon d'interaction défini autour de chaque élément, sera en interaction avec celui-ci. Cette définition d'un rayon d'interaction est une des différences existant entre ce modèle et celui des éléments distincts. L'interaction entre deux éléments se fait sur la base d'une surface infiniment petite assimilable à un point. On considère des interactions dites « molles », ce qui implique que deux éléments peuvent s'interpénétrer ou se chevaucher, néanmoins la géométrie des éléments reste indéformable, c'est à dire sphérique.

La zone d'interpénétration est petite par rapport à la taille des éléments et ne représente que quelques pour-cent du rayon des éléments. L'interaction des éléments est traitée comme étant un processus dynamique qui atteint l'équilibre lorsque les forces internes sont à l'équilibre.

L'équation du mouvement relie le mouvement de chaque élément à la somme des forces appliquées à cet élément Le comportement dynamique du système est résolu numériquement par un algorithme en temps dans lequel on considère que les vitesses et les accélérations sont constantes à chaque pas de temps. Ceci est fait par le biais d'une méthode explicite de différences finies. Le pas de temps est choisi suffisamment petit, afin que la perturbation induite ne se propage qu'entre un élément donné et ses plus proches voisins. Ainsi, à chaque pas de temps, le bilan des forces pour un élément donné est calculé en fonction de toutes les forces appliquées à cet élément en prenant en compte tous les éléments avec lesquels celui-ci est en interaction. La résolution du problème passe alors par le calcul alterné de l'équation du mouvement pour un élément donné et du bilan des forces appliquées à cet élément.

Tout le développement qui suit sera fait pour le problème à trois dimensions et selon des notations utilisées par d'autres auteurs (Cundall, 1988, Hart et al., 1988).

Bilan des forces de déplacement

Considérons deux éléments discrets sphériques *E1* et *E2* de rayon R^{E1} et R^{E2} respectivement, dont les centres sont en x_i^{E1} et x_i^{E2}. Ces deux éléments peuvent interagir selon des lois d'interaction qui seront décrites plus loin.

Tout en notant que l'interaction entre deux éléments n'implique pas obligatoirement le contact entre ces deux éléments on peut définir un plan perpendiculaire à la direction normale séparant les deux centres que l'on appellera plan d'interaction et qui est défini par le vecteur unité n_i tel que :

$$n_i = \frac{x_i^{E2} - x_i^{E1}}{\left| x_i^{E2} - x_i^{E1} \right|} \tag{1}$$

La distance d'équilibre séparant les deux centres au moment t_c de la création de l'interaction est donnée par :

$$d_{eq} = \left| (x_i^{E2})^{t_c} - (x_i^{E1})^{t_c} \right| = \sqrt{\left[(x_i^{E2})^{t_c} - (x_i^{E1})^{t_c} \right]^2} \tag{2}$$

Appelons U^n le déplacement relatif du point d'application de l'interaction dans la direction normale et qui est donnée par :

$$U^n = d_{eq} - d. \tag{3}$$

où d est la distance d'équilibre séparant les deux centres au moment t de la détermination du déplacement.

La position du point d'application de l'interaction est :

$$x_i^{\text{int}} = x_i^{E1} + (R^{E1} - \tfrac{1}{2} U^n) n_i. \tag{4}$$

Le vecteur force d'interaction F_i, qui représente l'action de l'élément *E1* sur l'élément *E2* peut être décomposé en une composante normale F_i^n et une composante en cisaillement F_i^s par rapport au plan d'interaction :

$$F_i = F_i^n + F_i^s. \tag{5}$$

Le vecteur normal de la force d'interaction est calculé par :

$$F_i^n = K^n U^n n_i, \tag{6}$$

où K^n est la composante normale de la raideur au point d'application de l'interaction.

La composante en cisaillement de la force d'interaction est d'abord mise à jour en étant projetée dans le nouveau plan tangent tel que (Hart et al., 1988, Walton, 1993) :

$$\left\{F_s^i\right\}_{projetée} = F_i^s - e_{ijk}e_{kmn}F_j^s n_m^{ancien} n_n, \tag{7}$$

où n_m^{ancien} est l'ancien vecteur unitaire dans la direction normale au plan d'interaction et e_{ijk} est le symbole de permutation.

Cette nouvelle composante en cisaillement est ensuite projetée selon la nouvelle direction normale ce qui s'écrit (Potyondy, 1996) :

$$\left\{F_i^s\right\}_{mise\_à\_jour} = \left\{F_i^s\right\}_{projetée} - \left\{F_i^s\right\}_{projetée} e_{ijk}\overline{\omega}_k \Delta t, \tag{8}$$

où $\overline{\omega}_k$ est la vitesse angulaire moyenne des deux éléments par rapport à la nouvelle direction normale :

$$\overline{\omega}_i = \frac{1}{2}\left(\omega_j^{E1} + \omega_j^{E2}\right) n_j n_i, \tag{9}$$

où ω_j^{E1} est la vitesse rotationelle de l'élément *E1* et ω_j^{E2} est celle de l'élément *E2*.

Le mouvement relatif au point d'interaction, ou la vitesse d'interaction V_i, qui est définie comme étant la vitesse de l'élément *E1* par rapport à l'élément *E2*, est donnée par :

$$V_i = \left(\dot{x}_i^{\text{int}}\right)_{E2} - \left(\dot{x}_i^{\text{int}}\right)_{E1} = \left(\dot{x}_i^{E2} + e_{ijk}\omega_j^{E2}\left(\dot{x}_k^{\text{int}} - \dot{x}_k^{E2}\right)\right) - \left(\dot{x}_i^{E1} + e_{ijk}\omega_j^{E1}\left(\dot{x}_k^{\text{int}} - \dot{x}_k^{E1}\right)\right), \tag{10}$$

où $\dot{x}_i^{E2}$ et $\dot{x}_i^{E2}$ sont les vitesses de translation des éléments *E1* et *E2*.

La vitesse d'interaction peut être décomposée en une composante normale et une composante en cisaillement par rapport au plan d'interaction. Appelons ces composantes V_i^n et V_i^s, alors la composante en cisaillement peut s'écrire :

$$V_i^s = V_i - V_i^n = V_i - V_j n_j n_i. \tag{11}$$

La composante en cisaillement du vecteur incrémental du déplacement au point d'interaction pendant un intervalle Δt peut se calculer grâce à :

$$\Delta U_i^s = V_i^s \Delta t, \tag{12}$$

et est utilisée pour calculer le vecteur incrémental de la composante en cisaillement de la force élastique (Hart et al., 1988) :

$$\Delta F_i^s = -k^s \Delta U_i^s, \tag{13}$$

où k^s est la composante en cisaillement de la raideur au point d'interaction. La valeur de k^s est déterminée à l'aide du modèle courant de raideur pour l'interaction.

La nouvelle force d'interaction en cisaillement se calcule en additionnant l'ancien vecteur force de cisaillement qui existait au début de l'intervalle de temps (après projection pour prendre en compte le mouvement du plan d'interaction) au vecteur incrémental de la composante en cisaillement de la force élastique soit :

$$F_i^s = \left\{F_i^s\right\}_{mise\_à\_jour} + \Delta F_i^s. \tag{14}$$

La contribution de la force totale d'interaction à la force résultante et aux moments appliqués aux deux éléments en interaction est donnée par :

$$\begin{aligned} F_i^{E1} &:= F_i^{E1} - F_i \\ F_i^{E2} &:= F_i^{E2} + F_i \\ M_i^{E1} &:= M_i^{E1} - e_{ijk}\left(x_j^{\text{int}} - x_j^{E1}\right)F_k \\ M_i^{E2} &:= M_i^{E2} + e_{ijk}\left(x_j^{\text{int}} - x_j^{E2}\right)F_k \end{aligned} \tag{15}$$

Où F_i^{E1} , F_i^{E2} sont les bilans des forces et M_i^{E1} et M_i^{E2} sont les bilans des moments des éléments *E1* et *E2* et F_i sont les forces calculées avec l'équation (5).

Equation du mouvement

Le mouvement d'un seul élément est calculé à partir de la force résultante et des vecteurs moment qui lui sont appliqués. Il peut être décrit en termes du mouvement en translation et du mouvement en rotation.

Le mouvement en translation du centre de masse de l'élément est décrit en fonction de sa position x_i, de sa vitesse $\dot{x}_i$, et de son accélération $\ddot{x}_i$; le mouvement en rotation de l'élément est décrit en termes de sa vitesse angulaire ω_i et de son accélération angulaire $\dot{\omega}_i$.

Les équations du mouvement s'écrivent à l'aide de deux équations vectorielles, la première étant le rapport entre la force résultante et le mouvement en translation tandis que la seconde donne le rapport entre le moment résultant et le mouvement en rotation.

L'équation du mouvement en translation peut s'écrire sous forme vectorielle :

$$F_i = m\ddot{x}_i, \tag{16}$$

où F_i est la force résultante qui est la somme de toutes les forces externes appliquées à l'élément (dont la force gravitationnelle) et m est la masse totale de l'élément.

L'équation du mouvement en rotation peut s'écrire :

$$M_i = \dot{H}_i \tag{17}$$

où M_i est le moment résultant appliqué à l'élément et $\dot{H}_i$ est le moment angulaire de l'élément. Pour un élément sphérique de rayon R dont la masse est distribuée uniformément dans son volume, le centre de masse coïncide avec le centre de la sphère. Alors tout axe de coordonnées locales est un axe principal et les moments inertiels dans chaque axes sont égaux.

Donc pour un élément sphérique le mouvement en rotation peut s'écrire,

$$M_i = I\dot{\omega}_i = \left(\frac{2}{5}mR^2\right)\dot{\omega}_i. \tag{18}$$

Les équations du mouvement (16) et (18) sont intégrées à l'aide d'un schéma centré en différences finies dont le pas de temps est Δt. Celui utilisé ici est de type « leap-frog » (Allen & Tildesley, 1987). Les quantités $\dot{x}_i$ et ω_i se calculent aux temps intermédiaires $t \pm n\,\Delta t/2$, tandis que les quantités x_i, $\ddot{x}_i$, $\dot{\omega}_i$, Fi et Mi sont calculés aux temps $\Delta t \pm n\,\Delta t$.

Les accélérations en translation et en rotation au temps t en termes des vitesses aux temps intermédiaires sont,

$$\begin{aligned}\ddot{x}_i^{(t)} &= \frac{1}{\Delta t}\left(\dot{x}_i^{(t+\Delta t/2)} - \dot{x}_i^{(t-\Delta t/2)}\right),\\ \dot{\omega}_i^{(t)} &= \frac{1}{\Delta t}\left(\omega_i^{(t+\Delta t/2)} - \omega_i^{(t-\Delta t/2)}\right).\end{aligned} \tag{19}$$

En remplaçant ces termes dans les équations 16 et 18 et en résolvant les vitesses au temps $(t+\Delta t/2)$ on obtient :

$$\dot{x}_i^{(t+\Delta t/2)} = \dot{x}_i^{(t-\Delta t/2)} + \left(\frac{F_i^{(t)}}{m}\right)\Delta t,$$
$$\omega_i^{(t+\Delta t/2)} = \omega_i^{(t-\Delta t/2)} + \left(\frac{M_i^{(t)}}{m}\right)\Delta t. \qquad (20)$$

Enfin les vitesses de l'équation (20) sont utilisées pour mettre à jour les positions des centres des éléments (Hart et al., 1988) :

$$x_i^{(t+\Delta t/2)} = x_i^{(t)} + \dot{x}_i^{(t+\Delta t/2)}\Delta t. \qquad (21)$$

Les valeurs de $F_i^{(t+\Delta t)}$ et de $M_i^{(t+\Delta t)}$ qui seront utilisées au cycle suivant sont obtenues en appliquant le bilan des forces de déplacement.

Modèles d'interaction

Le comportement global d'un matériau est reproduit dans *SDEC V 2.0* en associant un modèle de comportement simple à chaque interaction. Rappelons qu'une interaction se fait dans un rayon d'interaction et n'implique pas forcément que deux éléments se touchent.

Si la force d'interaction entre ces deux éléments est de nature cohésive, on peut alors introduire la notion de rupture si le seuil de la force appliquée dépasse le seuil maximal de résistance en tension T (avec $T>0$) entre ces deux éléments :

$$F^n = K^n U^n < -A_{\text{int}} T, \qquad (22)$$

où A_{int} est la surface moyenne où s'exerce l'interaction et qui est définie par,

$$A_{\text{int}} = \pi \min(\left(R^{E1}\right)^2, \left(R^{E2}\right)^2). \qquad (23)$$

Suite à une rupture, les composantes normales et de cisaillement de la force d'interaction deviennent alors nulles. Sinon la force de cisaillement maximale est calculée par :

$$F^s_{\max} = cA_{\text{int}} + F^n \tan\phi_{\text{int}}, \qquad (24)$$

où c est la cohésion et ϕ_{int} est l'angle de frottement pour les interactions cohésives. Si la valeur absolue de la force de cisaillement est telle que :

$$F^s = \left(F_i^s F_i^s\right)^2 > \left|F^s_{\max}\right|, \qquad (25)$$

alors la force de cisaillement est réduite à :

$$F_i^s = F_i^s \left[F_{\max}^s / F^s \right] \tag{26}$$

Afin de reproduire la nature irréversible de la fracturation on admet que toute nouvelle interaction qui se produira lors d'un contact c'est à dire quand, $\left(\left| x_i^{E2} - x_i^{E2} \right| \right) < (R^{E1} + R^{E2})$, sera uniquement de nature répulsive, c'est à dire $T=0$ et $c=0$.

La force normale durant ce contact est donnée par (Walton, 1993) :

$$F_i^n = \begin{cases} K^n U^n n_i & \text{pendant le chargement,} \\ \zeta K^n \left(U^n - U_0^n \right) n_i & \text{pendant le déchargement,} \end{cases} \tag{27}$$

où $\zeta \geq 1$ est relié au coefficient de restitution e par,

$$\zeta = \frac{1}{e^2}. \tag{28}$$

U_0^n représente le chevauchement relatif quand la force de déchargement est nulle (ceci pour tenir compte de la déformation inélastique de la surface de contact).

Modèle linéaire d'interaction

Les raideurs qui, au moment de l'interaction, relient les forces d'interaction avec les déplacements relatifs dans les directions normales et de cisaillement par le biais des équations (6) et (13) suivent un modèle linéaire.

On suppose que les raideurs normales et de cisaillement K^n et k^s des deux éléments en interaction agissent en série. La raideur normale d'interaction est donnée par :

$$K^n = 2 \left[\frac{k_n^{E1} k_n^{E2}}{k_n^{E1} + k_n^{E2}} \right], \tag{29}$$

et la raideur en cisaillement est :

$$k^s = 2 \left[\frac{k_s^{E1} k_s^{E2}}{k_s^{E1} + k_s^{E2}} \right], \tag{30}$$

où E1 et E2 sont les deux éléments qui interagissent.

Homogénéisation

Les techniques d'homogénéisation consistent à obtenir les équations macroscopiques du comportement d'un matériau donné, à partir de son comportement et de sa structure microscopique (Cambou et al., 1995). Dans le cas présent, le matériau est représenté par

un modèle numérique d'éléments discrets et il faut donner les relations entre les propriétés élastiques, ici le coefficient de Poisson et le module d'Young, que l'on veut reproduire et les constantes locales de raideur K^n et k^s.

Pour cela, des tests numériques en compression uniaxiale sont effectués, dans lesquels aucune rupture n'est permise (le seuil de rupture est artificiellement élevé) et pour lesquels on fait varier le rapport k^s / K^n. Ceci permet d'obtenir les courbes reproduites dans la figure 1.

Les équations obtenues par approximation sont données par :

$$E = D_l \frac{D_{eq}}{A_{\text{int}}} k^n \left(\frac{0.825k^n + 2.65k^s}{2.5k^n + k^s} \right), \tag{31}$$

et,

$$\nu = \frac{k^n - k^s}{2.5k^n + k^s}, \tag{32}$$

où D_l est le densité de liaisons.

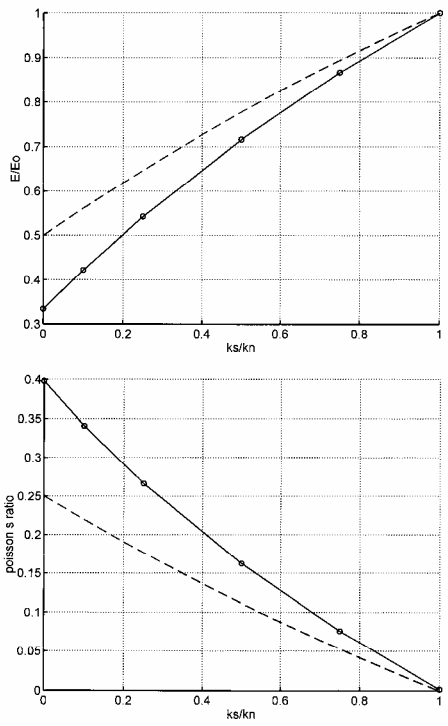


Figure 1. En haut, la variation du module d'Young et en bas, la variation du coefficient de Poisson. En trait plein, la représentation des équation 31 et 32. En pointillé, le modèle de Voigt.

Détection des interactions

Une des parties principales du code *SDEC* consiste à déterminer la proximité des éléments les uns par rapport aux autres. Puis, lorsque ce voisinage a été établi, il s'agit de déterminer le plus rapidement possible s'ils sont en interaction pour pouvoir ensuite calculer les forces d'interaction. Ceci peut être fait de plusieurs façons (Discrete Element Project, 1997) dont deux ont été sélectionnées pour *SDEC V 2.0*, la première étant une méthode dite de grille et la deuxième une méthode dite de tri par empilement.

Génération et répartition des éléments

Lors de précédents travaux (Donzé & Magnier, 1995, Donzé et al. 1997, Magnier & Donzé, 1997) l'importance de l'utilisation de milieux isotropes à la fracturation avait été mise en évidence. L'objectif était de pouvoir générer un ensemble compact d'éléments discrets de tailles différentes dans un volume 3D à géométrie complexe. Dans cette optique, *SDEC V1.0* génère le désordre en plusieurs étapes.

Premièrement, les éléments qui composent le modèle de départ, ont, tous un rayon identique R_{init}, et sont répartis selon un agencement régulier le plus compact possible, en CFC par exemple dans le cas tridimensionnel (en triangulaire compact en 2D).

La deuxième étape consiste à diminuer la taille des rayons des éléments par un facteur de correction, tel que,

$$R = \alpha_{correct} R_{init} \qquad (33)$$

avec $\alpha_{correct}=0.25$ dans le cas tridimensionnel.

Puis la position initiale des éléments est elle aussi modifiée de façon aléatoire en évitant le moindre chevauchement entre éléments (Donzé et al., 1997).

Le désordre ainsi créé implique qu'aucun élément discret n'est en contact. Il faut donc combler au maximum ce vide pour obtenir un milieu compact. Pour ce faire on va faire les éléments sont grossis. L'incrément de grossissement est identique à chaque itération. Ce grossissement des éléments se poursuit jusqu'à ce que, soit la taille maximum de grossissement permise soit atteinte, soit que le critère maximum de chevauchement avec un autre élément soit atteint. Ce dernier critère limite le chevauchement entre éléments à quelques pour-cent du rayon et une fois atteint, l'éléments grossissant voit sa taille ajustée afin d'assurer un simple contact sans chevauchement.

Lorsque tous les éléments sélectionnés ont atteint l'un de ces deux seuils, on les empêche de bouger dans l'étape suivante où il s'agit de faire grossir les éléments restants.

Ces derniers vont eux aussi grossir soit jusqu'à leur taille maximum, soit jusqu'à ce qu'ils aient atteint le critère maximum d'interpénétration.

Avec cette méthode, on obtient en moyenne 6 contacts par élément discret à 3D. Dans tous les cas de figure, on a au minimum un contact.

Figure 9. Proposition d'agencement des éléments discrets formant la colonne explosive. Ceux-ci sont représentés en noir.

3. Bibliographie

Allen, M. P., & D. J. Tildesley, *Computer Simulation of Liquids*, Clarendon Press, Oxford, 1987.

Cambou, B., P. Dubujet, F. Emeriault & F. Sidoroff, Homogenization for granular materials, *Eur. J. Mech. Solids*, **14**, 255-276, 1995.

Cundall, P. A., A computer model for simulating progresive large scale movements in blocky rock systems, *Proceedings of the Symposium of the International Society of Rock Mechanics*, Nancy, France, **1**, 1971.

Cundall, P. A., Distinct element models of rock and soil structure, in *Analytical and Computational Methods in Engineering Rock Mechanics*, E. T. Brown Ed., Londres, 129-163, 1987

Cundall, P. A., Formulation of a three dimensional distinct element model-Part I. A scheme to detect and represent contacts in a system composed of many polyhedral blocks, *Int. J. Rock Mech. Min. Sci. & Geomech. Abstr*., **25**, 107-116, 1988.

Cundall, P. A. & O. D. L. Strack, A discrete numerical model for granular assemblies, *Géotechnique*, **29**, 47-65, 1979.

Discrete Element Project (Magnier, S. A., & F. V. Donzé), Rapport N°2, Université du Québec à Monréal, 1997.

Donzé, F., & S. A. Magnier, Formulation of a three-dimensional numerical model of brittle behavior, *Geophys. J. Int*., **122**, 790-802, 1995.

Donzé, F. V., J. Bouchez, & S. A. Magnier, Modeling fractures in rock blasting, *Int. J. Rock Mech. Min. Sci*., 1997.

Hart, R., P. A. Cundall, & J. Lemos, Formulation of a three dimensional distinct element model- Part II. Mechanical calculations for motion and interaction of a system composed of many polyhedral blocks, *Int. J. Rock Mech. Min. Sci. \& Geomech. Abstr.,* **25**, 117-125, 1988.

Magnier, S. A., & F. V. Donzé, Numerical simulations of impacts using a discrete element method, *Mechanics of Cohesive-Frictional Materials*, 1997.

Müller, D., *Techniques Informatiques Efficaces Pour la Simulation de Milieux Granulaires par des Méthodes d'Eléments Distincts*, Thèse de doctorat, EPFL, 1996.

Potyondy, D., Personnal communication, 1996.

Walton, O. R., Numerical simulation of inclined chute flows of monodisperse, inelastic, frictional spheres, *Mechanics of Materials* , **16**, 239-247, 1993.

YSO, Personnal communication, 1999.

ISRN GEONUM-NST--2001-03--FR, 2001.
F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2.
GEOTOP, Université du Québec à Montréal, 1997.

PARTIE 2
Support de développement au logiciel SDEC

Sommaire

***ISRN GEONUM-NST--2001-03--FR*, 2001.**
F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2.
GEOTOP, Université du Québec à Montréal, 1997.

Introduction

Ce document est le complément technique du manuel d'utilisation *on line* fourni avec le logiciel SDEC[1]. Il contient une description détaillée des classes composant le logiciel.
Le contenu des fichiers sources est présenté et la progression logique des différents exécutables (*generate.exe*, *modify.exe* et *simulation.exe*) composant le logiciel SDEC est décomposée.

Les composants de l'environnement Windows ne sont toutefois pas détaillés, car ils ne représentent pas une partie essentielle du logiciel et ceux-ci peuvent être ignorés, même à l'utilisation.

Le logiciel SDEC est écrit en C++, certaines notations spécifiques à ce langage ont été conservées.

- la fonction fn(…) de la classe **Cfn**, sera déclarée comme étant la fonction **Cfn**::fn(…).
- un objet Obj d'une classe **Cobj** est créé, ceci est noté **Cobj** Obj.

Les noms de fichiers sont donnés en italiques.

Les librairies dynamiques notées **dll**, accessibles à l'utilisateur sont détaillées.

En dernière partie, des diagrammes font un bilan récapitulatifs des principales classes présentes dans les trois applications composant le logiciel SDEC

[1] SDEC est référencé par Frédéric Victor Donzé à l'INTERDEPOSIT, Fédération Internationale de l'Informatique et des Technologies de l'Information, sous le numéro IDDN.FR.010.0078044.000.R.P.2000.030.30000.

F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2. GEOTOP, Université du Québec à Montréal, 1997.

Création des éléments discrets : *generate.exe*

Cet exécutable permet de créer l'ensemble des éléments discrets qui serviront dans l'étape de simulation. Les fichiers sources composant cet exécutable, sont présentés à la figure 1.

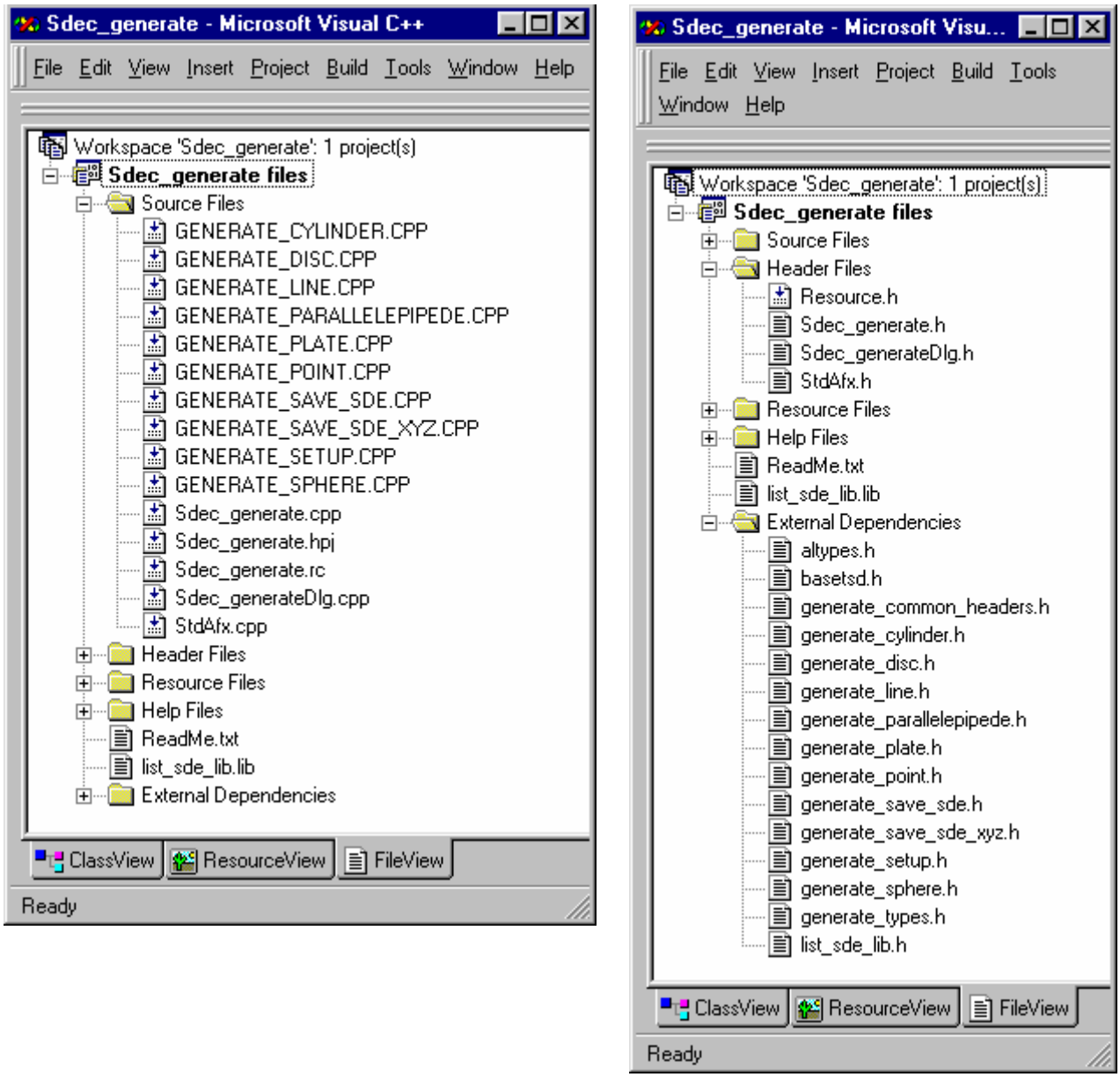


Figure 1. A gauche, les fichiers sources, à droite les fichiers entêtes qui leur sont associés.

Quand *generate.exe* est cliqué sur le menu principal de l'interface Windows du logiciel SDEC, la fenêtre de la figure 2 apparaît.

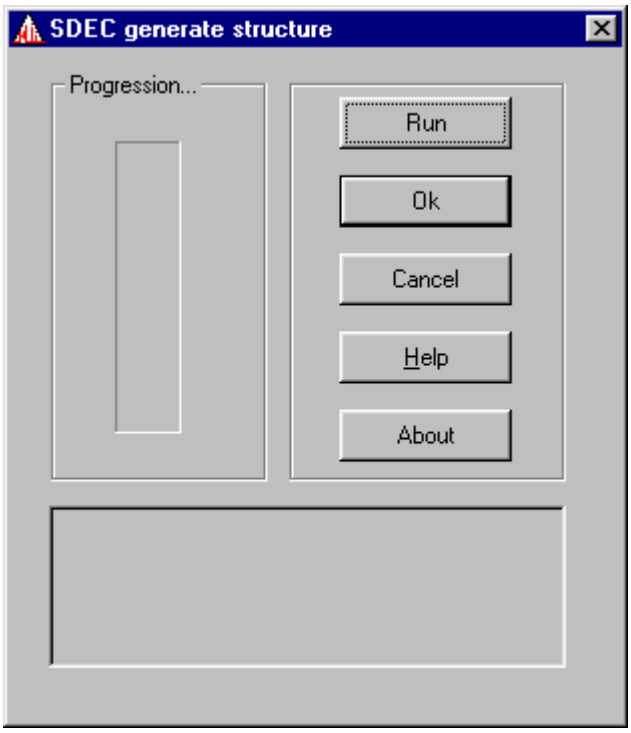


Figure 2. fenêtre de generate.exe

Une fois le bouton *Run* cliqué, la fonction **CSdec_generateDlg**::OnRun() du fichier source *Sdec_generateDlg.cpp* est exécutée. Les différentes étapes contenues dans cette fonction sont indiquées d'abord, ensuite leur détail est donné.

1. Un objet "liste d'éléments discrets" **SDEdllist** list_SDE est d'abord créé. Cet objet est défini dans le fichier *list_sde_lib.h*.

2. Ensuite l'objet **G_Setup** GSP(list_SDE, pt_SDE) est créé (classe **G_Setup** définie dans generate_setup.h). C'est cet objet qui va remplir la liste des éléments discrets en fonction des informations contenues dans les fichiers data remplis par l'utilisateur. Les opérations engendrées par la création de cet objet sont indiquées plus loin.

3. Après cela, l'objet **Save_SDE** SSDE(list_SDE, pt_SDE, GSP) déclaré dans *generate_save_sde.h* est créé. Il va permettre de sauvegarder la liste des éléments discrets dans un fichier sous format binaire, fichier qui sera utilisé par *modify.exe* ou *simulation.exe*.

4. Enfin, l'objet **Save_SDE_xyz** SSDE_XYZ(list_SDE, pt_SDE, GSP) déclaré dans *generate_save_sde_xyz.h* est créé et il va permettre de sauvegarder certains champs choisis de la liste des éléments discrets (par exemple, leurs tailles, leurs positions) dans

un fichier sous format ASCII pour permettre de visualiser l'ensemble des éléments discrets créés avec un logiciel de représentation graphique.

Détail de l'étape de création des groupes (ou structures) d'éléments discrets

Objet **G_Setup** GSP(list_SDE, pt_SDE) :

A sa création, le constructeur de l'objet **G_Setup** GSP(list_SDE, pt_SDE) est exécuté. Il va d'abord lire de façon séquentielle le fichier *setup_generate.sim* qui contient les noms des fichiers "structure.str" et "medium.med" à ouvrir. Ces fichiers contiennent les informations des différents groupes d'éléments discrets à créer ainsi que les propriétés qui leur seront associées (et qui sont lus au niveau des fonctions **G_Setup**::Read_structure_file(…) et **G_Setup**::Read_medium_file(…)). En fonction des mots clefs rencontrés dans les fichiers "structure.str" (mots clefs : Plate, Parallelepipede, Cylinder, Sphere, Disc, Line, Point) , les objets correspondants sont créés dynamiquement.

Exemple : à la lecture du mot clef "Plate", l'objet **Plate** *obj_plate (classe définie dans le fichier source *generate_plate.h)* est créé dynamiquement. Suite à sa création, la fonction **Plate**::Setup(…) (dans le fichier source *generate_plate.cpp*) est exécutée. Cette fonction fait appel de façon séquentielle aux fonctions suivantes :

- **Plate**::SetLattice(…); affectation des positions de tous les éléments discrets
- **Plate**::Adjust_borders(…); élimination des éléments discrets créés hors des limites géométriques.
- **Plate**::Centring_geometry(…); recentrage des éléments discrets autour de l'origine.
- **Plate**::SetProperties(…); affectation des propriétés issues du fichier "medium.med" pour chaque élément discret.
- **Plate**::Adjust_geometry(…); fonction ne faisant rien…
- **Plate**::Rotate(…); rotation de l'ensemble des éléments discrets autour d'un axe défini dans le fichier "structure.str".
- **Plate**::Translate(…); translation des positions des éléments discrets suivant la position globale de la structure donnée dans le fichier "structure.str".

Toutes ces fonctions sont dans le fichier source *generate_plate.cpp.*

Si, à la place du mot clef "Plate", un autre mot clef eut été lu, l'ensemble des éléments discrets correspondant à cette structure eut été élaboré de la même façon. Par exemple, pour le mot clef "Cylinder", l'objet **Cyl** *obj_cyl aurait été créé, et la fonction **Cyl**::Setup(…) appelée.
Note, afin de ne pas réécrire les fonctions utilisées communément par les différentes classe de structure, la classe Plate est parente aux autres classes (c-a-d qu'elles en héritent) et seules les fonctions ayant un contenu différent sont surchargées (c-a-d réécrites mais en gardant le même nom).

Modifications dans la liste des éléments discrets : *modify.exe*

Cet exécutable permet de lire les fichiers binaires contenant la liste des éléments discrets et créés par *generate.exe* ou *simulation.exe*.
Les fichiers sources composant cet exécutable, sont présentés à la figure 3.

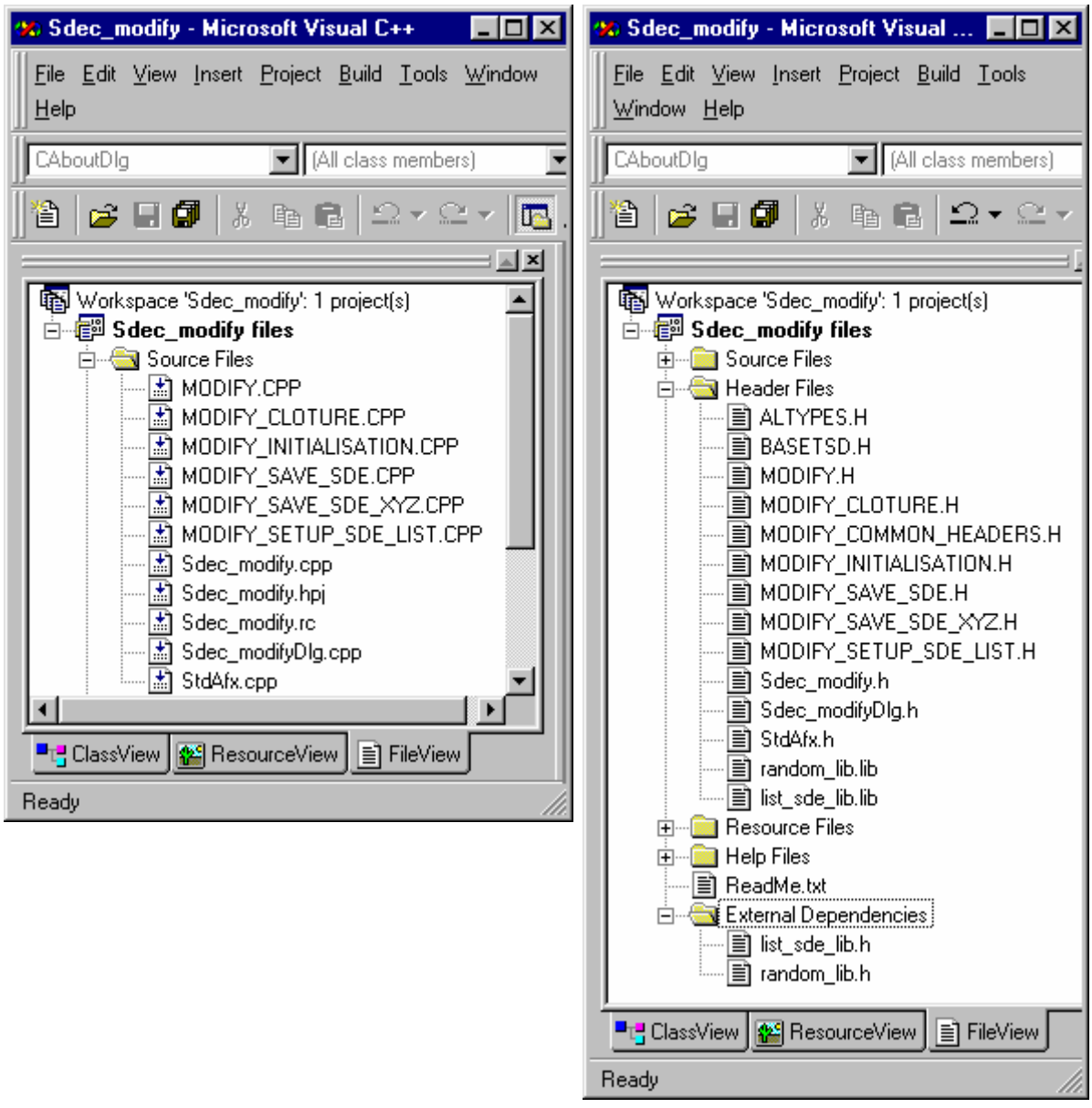


Figure 3. A gauche, les fichiers sources de modify.exe, à droite les fichiers entêtes qui leur sont associés.

Quand *modify.exe* est cliqué sur le menu principal de l'interface Windows du logiciel SDEC, la fenêtre de la figure 4 apparaît.

Figure 4. fenêtre de modify.exe

Une fois le bouton *Run* cliqué, la fonction **CSdec_modifyDlg::**OnButtonRun() du fichier source *Sdec_modifyDlg.cpp* est exécutée. La partie principale de cette fonction est la création dynamique de l'objet **Cmodify** *MODIFY_OBJ, dont la classe est définie dans le fichier *modify.h.*

Détail de l'étape de la modification dans la liste des éléments discrets

A la création de l'objet MODIFY_OBJ, la fonction **Init_modif**::Init_modif() (dans *modify_initialisation.cpp*) est d'abord activée. Cette fonction va récupérer la liste des éléments discrets contenus dans le fichier binaire issu de *generate.exe* ou de *simulation.exe*, en créant dynamiquement l'objet **M_Setup** *MSP, défini dans *modify_setup_sde_list.h*.
A sa création, cet objet ouvre le fichier paramètre *setup_modify.sim*. avec la fonction **M_Setup**::Read_DEW_file(…) contenu dans *modify_setup_sde_list.cpp*, le fichier binaire contenant les caractéristiques des éléments discrets est lu et la liste dynamique des éléments discrets utilisée dans *modify.exe* est chargée.

La fonction **Cmodify**::init_dll_specific() (dans *modify.cpp*) est ensuite appelée. Cette fonction va ouvrir la dll spécifique (voir chapitre sur la dll spécifique de modification)

remplie par l'utilisateur dans le fichier de données dont le nom est récupéré par la fonction **M_Setup**::Read_DEW_file(…).

L'étape de modification dont les spécificités sont contenues dans la dll de l'utilisateur est ensuite réalisée par le biais de la fonction **Cmodify**::dll_load() (dans *modify.cpp*).
Note : si des éléments discrets sont retirés de la liste, leur destruction physique se fait par la fonction **Cmodify**::delete_operation(), et non pas dans la dll, car le destructeur de la liste n'est pas exportable (liste créée dans modify.exe).

La fonction **Close_modif**::save_files(…) (classe définie dans *modify_cloture.h*) est enfin appelée pour sauvegarder les différents fichiers. (Noter que cette fonction peut être directement appelée depuis le constructeur Cmodify() car sa classe hérite de la classe Close_modif). L'un des fichiers est binaire et contient les informations complètes des éléments discrets et il est réalisé par l'intermédiare de la création de l'objet **Save_SDE**::Save_SDE(…) (classe définie dans *save_sde.h*). L'autre fichier sous format ascii et contenant des informations sélectionnées pour être visualisées ou analysées, est réalisé par la création de l'objet **Save_SDE_xyz::**Save_SDE_xyz() (classe définie dans *modify_save_sde_xyz.h*).

Suite à cela, l'exécutable modify.exe a fini sa tache.

La dll spécifique de modification

La **dll** (dynamic linked librarie) spécifique de modification, est une librairie compilée à part. Cette librairie contient les aspects spécifiques de l'opération de modification à apporter dans la liste des éléments discrets et elle est remplie par l'utilisateur puis compilée par celui-ci. Le nom de la dll doit correspondre à celui entré dans le fichier de paramètres (*setup_modify.sim*) afin qu'elle puisse être appelée.

Cette dll se situe dans le répertoire spécifique au projet traité, exactement dans le répertoire suivant : *C:\myproject\dll\modify\modify_dll* si le projet traité s'intitule *myproject* et qu'il situé sur le volume *C:* .

Sur la figure 5, est représenté sous environnement visual, le projet de la dll de modification.

Le fichier source, appelé comme désiré par l'utilisateur, ici source_dll.cpp, contient la fonction suivante : MODIFY_DLL_API int fn_modify(…). C'est cette fonction avec ce nom là qui est exporté vers l'application midify.exe. C'est donc cette fonction qui doit contenir les instructions choisies pour modifier ou non, le contenu de la liste des éléments discrets.
Noter que la première partie de cette fonction sert à lire d'éventuels paramètres que l'on souhaite standardiser pour la dll. Ces paramètres sont contenus dans le fichier paramètre intitulé du nom de la dll suivi du suffixe .mod.

Généralement, le code généré par l'utilisateur doit se situer entre les bornes suivantes :

```
//+++++++++++++++++++++++++++++++++
// ADD YOUR CODE HERE
//+++++++++++++++++++++++++++++++++
```

et

```
//+++++++++++++++++++++++++++++++++
// END OF YOUR CODE HERE
//+++++++++++++++++++++++++++++++++
```

Tout autre ligne doit rester intacte.

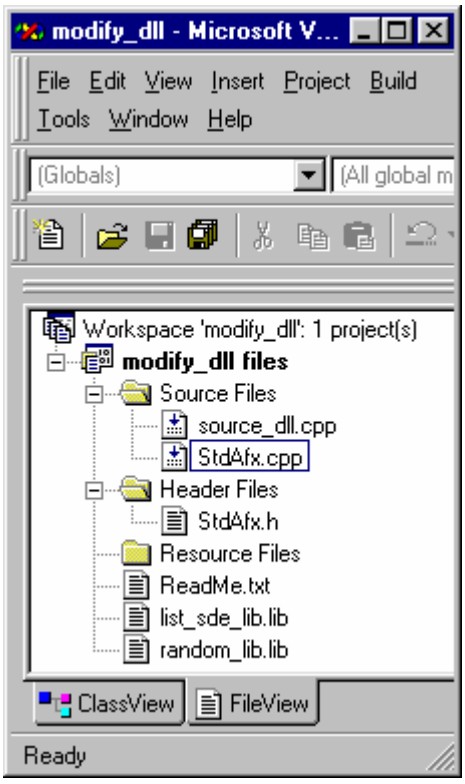


Figure 5. DLL de modification

***ISRN GEONUM-NST--2001-03--FR*, 2001.**
F. V. Donzé & S.-A. Magnier, « Spherical Discrete Element Code » In: *Discrete Element Project Report* no. 2.
GEOTOP, Université du Québec à Montréal, 1997.

Evolution du système d'éléments discrets : *simulation.exe*

Cet exécutable utilise les fichiers binaires contenant la liste des éléments discrets créés par *generate.exe* ou *modify.exe*.
Les fichiers sources composant cet exécutable, sont présentés à la figure 6.

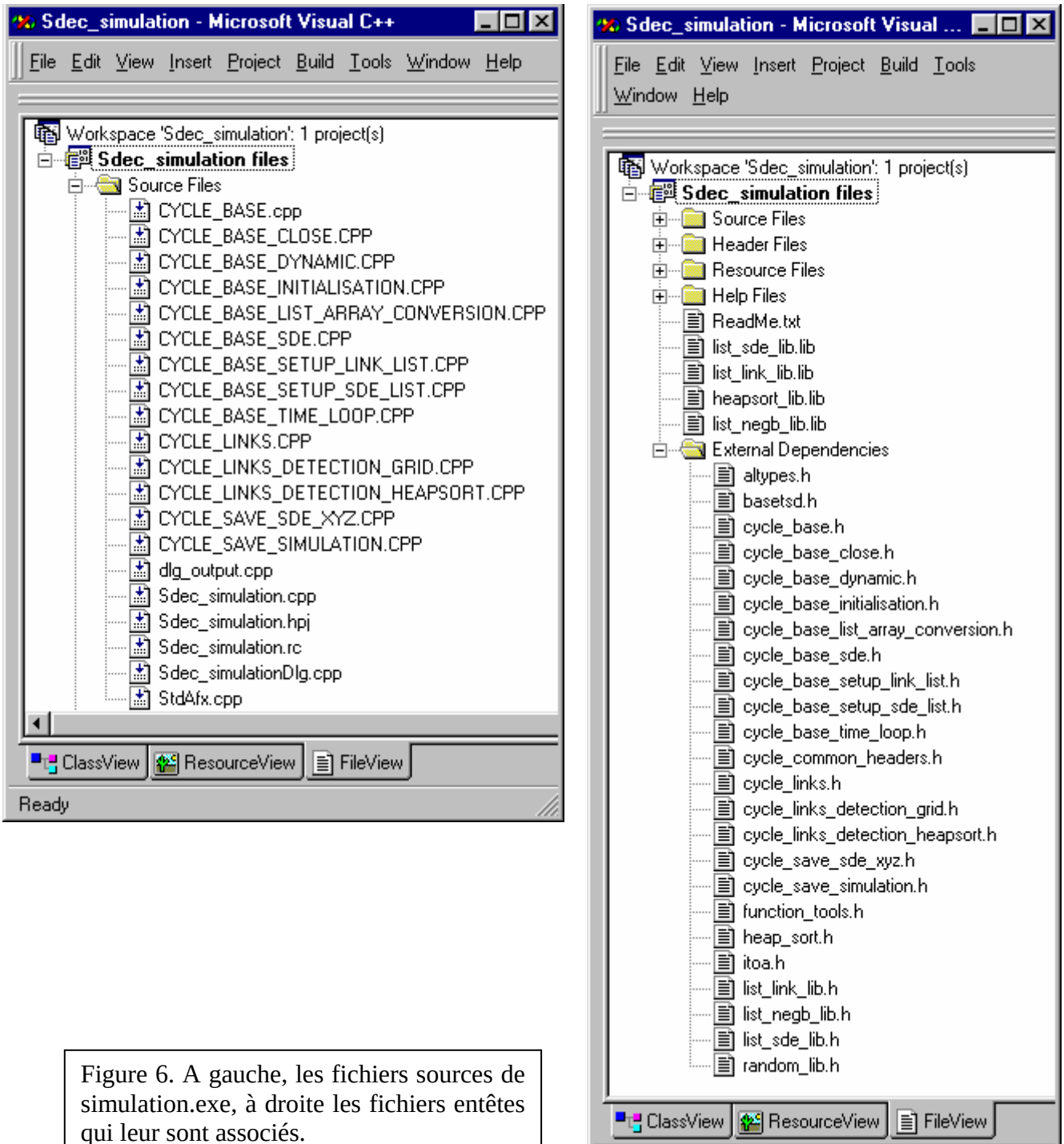


Figure 6. A gauche, les fichiers sources de simulation.exe, à droite les fichiers entêtes qui leur sont associés.

Quand *simulation.exe* est cliqué sur le menu principal de l'interface Windows du logiciel SDEC, la fenêtre de la figure 7 apparaît.

Figure 7. Fenêtre de simulation.exe

Une fois le bouton *Run* cliqué, la fonction **CSdec_simulationDlg::**OnButtonRun() du fichier source *Sdec_simulationDlg.cpp* est exécutée. La partie principale de cette fonction est la création dynamique de l'objet **Cycle** *Cycle_obj, dont la classe est définie dans le fichier *cycle_base.h.*

Détail de l'étape de simulation utilisant l'ensemble des éléments discrets

L'objet Cycle_obj est issu d'une classe (**Cycle**) qui hérite de plusieurs classes (**CDialog, Init_system, Time_loop, Close_system**). A sa création, les constructeurs des classes parentes vont entrer en action.
La classe **Cdialog** est la première à s'activer, mais son constructeur ne fait rien. Cette classe est là pour initialiser la fenêtre Windows sur laquelle sera affichée la progression du calcul.
La classe **Init_system** (définie dans *cycle_base_initialisation.h*) va initialiser le système des éléments discrets. Une fois le travail de ce constructeur achevé, le constructeur de la classe **Time_loop** (définie dans *cycle_base_time_loop.h*) va initialiser l'étape d'évolution du système des éléments discrets. Enfin, le constructeur de la classe **Close_system** (définie dans *cycle_base_close.h*) va initialiser les éléments nécessaires à clôturer le système.

Détail de l'initialisation du système

A) La première fonction à entrer en action est **Init_system**::Init_system(), présente dans le fichier *cycle_base_initialisation.cpp.*

1. Cette fonction crée d'abord dynamiquement l'objet DEG = new **DEW_Get**(…) qui va mettre en place la liste des éléments discrets. Pour cela, le constructeur **DEW_Get**::DEW_Get() (définie dans *cycle_base_setup_sde_list.h*) entre en jeu (fichier *cycle_base_setup_sde_list.cpp*). Cette fonction lit les fichiers paramètres globaux, contenus dans *setup_cycle.sim* et les paramètres spécifique contenus dans le fichier *specific_file.clc* défini dans *setup_cycle.sim*. La fonction **DEW_Get**::Read_DEW_file(…) est appelée dans le constructeur et c'est elle qui va faire la lecture du fichier du fichier binaire où se trouvent stockées les informations des éléments discrets, pour les rentrer dans la liste chaînée, list_SDE.

2. Ensuite cette fonction crée dynamiquement l'objet DE = new **SDE** […], objet tableau dont la classe est définie dans *cycle_base_sde.h*, et dont le constructeur SDE::SDE() initialise tous les éléments du tableaux qui recevront les informations propres à chaque élément discret (et provenant de la liste list_SDE). Cet objet tableau sera utilisé pour traiter les éléments discrets plus rapidement que cela pourrait être fait avec le liste chaînée, lors de l'évolution du système.

3. Suite à cela, l'objet dynamique Links_properties = new **Links**(…) est crée. La classe **Links** est déclaré dans le fichier *cycle_links.h* et définie dans le fichier *cycle_links.cpp*. Le constructeur de cet objet initialise toutes les fonctions qui traitent de l'état des interactions via la **dll** spécifique aux interactions utilisées.

4. L'objet LA = new **List_Array**(…) va réaliser le transfert des éléments discrets de la liste chaînée list_SDE , remplie à l'étape 1, au tableau DE. La classe **List_Array** est définie dans *cycle_base_list_Array_conversion.h*, et c'est dans le fichier *cycle_base_list_Array_conversion.cpp* que le constructeur **List_Array**:: List_Array (…) appelle différentes fonctions pour faire le transfert. Note, Si les données des éléments discrets sont directement issues d'un fichier binaire provenant d'une précédente simulation (soit générée par simulation.exe), c'est ici que ces données sont lues (y compris les informations concernant les voisins de chaque élément discret). Cette information est lue depuis le fichier spécifique.clc.

5. Enfin, l'objet LS = new **Link_setup** va initialiser la liste des liaisons qui seront représentatives des interactions entre les éléments discrets. La classe **Link_setup** est définie dans *cycle_base_setup_link_list.h*. Le constructeur **Link_setup**::Link_setup(…) (dans le fichier *cycle_base_setup_link_list.cpp*), va charger la liste des liaisons. Note, il peut charger un fichier contenant des liaisons existantes entre les différents éléments discrets (en accord avec la note du point 4), si cela est indiqué dans le fichier spécifique.clc (ceci est réalisé par la fonction **Link_setup**::Read_Link_file()). Dans tous les cas, le constructeur fait appel à la fonction **Link_setup**::Link_creation, qui va stocker les liaisons si celles-ci doivent être créées (dans le cas de la lecture de fichier, les liaisons déjà stockées dans la liste chaînée seront vérifiées). Pour ce faire, l'algorithme de reconnaissance des voisins est initialisé, suivant le choix de l'algorithme, soit par découpage de l'espace (grille spatiale) soit par tri. Par grille spatiale, l'objet DETECT_LINK_BY_GRID = new **Grid**(…) (classe définie dans *cycle_links_detection_grid.h*) dont le constructeur va faire appel à la fonction Grid::Initialisation_Grid() (dans le fichier *cycle_links_detection_grid.cpp*) pour l'initialisation est créé. Ensuite les fonction DETECT_LINK_BY_GRID->LINK_INIT() ou DETECT_LINK_BY_GRID->LINK_RT_fixe_init() seront appelées pour reconnaître les liaisons à stocker dans la chaîne liée list_LINK, suivant le régime de recherche des contacts pendant le déroulement de la simulation. La démarche est semblable dans le cas du choix de détection par tri. L'initialisation se fait en créant l'objet DETECT_LINK_BY_HEAPSORT = new **Heap_SRT**(…)(classe définie dans *cycle_links_detection_heapsort.h)*

Plus de détails sont donnés sur la gestion des interactions dans le chapitre : **Détermination des interactions : création des liaisons**

B) La seconde fonction à entrer en action est **Dynamic_cycle::**Dynamic_cycle(…) présente dans le fichier *cycle_base_dynamic.cpp.* Cette fonction a pour vocation d'initialiser les variables qui vont être nécessaires aux calculs des forces durant la phase de simulation. Elle définie aussi le nombre d'itérations et le style d'intégration utilisés.

Ces deux premières étapes font parties des actions engendrées par les différents constructeurs des classes parentes à la classe Cycle. A partir de là, c'est le constructeur de la classe Cycle qui entre en action.

C1) La première fonction appelée dans le constructeur de **Cycle** est **Cycle::**init_output_dlgbox() qui se trouve dans le fichier *cycle_base.cpp*.. Cette fonction initialise la boite de dialogue sans mode (définie dans *dlg_output.h*) qui va afficher la valeur de l'itération traitée par l'exécutable *simulation.exe*.

C2) La deuxième fonction est **Cycle**::init_dll_specific() qui se trouve dans le fichier *cycle_base.cpp*. Cette fonction va connecter les **dll** spécifiques associées à l'exécutable *simulation.exe*. Elle va aussi initialiser les fonctions spécifiques définies dans ces **dll** et qui vont être utilisées dans l'application principale *simulation.exe.*

C3) La fonction suivante, read_specific_data_file(), est une fonction définie dans la dll spécifique principale. Elle va lire dans le fichier de paramètres spécifiques à la dll, les différents paramètres dont la dll a besoin pour fonctionner suivant la spécificité donnée du problème.

C4) Ensuite, la fonction, init_specific_array_tool(), elle aussi définie dans la dll spécifique, initialise les tableaux et variables utilisés dans la dll.

Détail sur l'évolution du système

C5) Suit la fonction **Time_loop**::run_cycle (déclarée dans le fichier *cycle_base_time_loop.h* et définie dans *cycle_base_time_loop.cpp*) qui est une fonction clef de l'application. L'exécution de cette fonction engendre l'évolution du système d'éléments discrets, dans l'ordre suivant :

do {

//détection des nouveaux contacts au temps t (fonction définie dans
//*cycle_base_time_loop.h* mais qui fait appel aux fonctions de détermination des
//nouveaux contacts définies soit dans *cycle_links_detection_grid.cpp*, soit dans
//*cycle_links_detection_heapsort.cpp*, suivant le choix exprimé dans les fichiers
//paramètres) :

check_new_contact(…);

//détermination du pas en temps, fonction définie *cycle_base_time_loop.h* mais qui //fait appel aux fonctions soit **Dynamic_cycle::**Timestep_scaled() soit //**Dynamic_cycle::**Timestep() définies dans le fichier *cycle_base_dynamic.cpp* :

check_Timestep(…);

//détermination de la force d'interaction au temps t, en appelant la fonction //**Dynamic_cycle::**Force_determination() définie dans le fichier //*cycle_base_dynamic.cpp* :

Force_determination();

//détermination des conditions imposées aux forces calculées au temps t en //appelant la fonction vide **Time_loop::**perturbation_Force() définie dans //*cycle_base_time_loop.h*, mais surchargée par celle définie dans la dll spécifique :

perturbation_Force(…);

//détermination de la solution pour les vitesses et les positions des éléments discrets //au temps (t+dt/2), en appelant la fonction **Dynamic_cycle::**Solution(), présente //dans le fichier *cycle_base_dynamic.cpp*.

Solution(…);

// détermination des conditions imposées aux vitesses calculées au temps t+dt/2 en //appelant la fonction vide **Time_loop::**perturbation_velocity() définie dans //*cycle_base_time_loop.h*, mais surchargée par celle définie dans la dll spécifique :

perturbation_velocity(…);

//détermination des conditions imposées aux déplacements calculés au temps t+dt
//en appelant la fonction vide **Time_loop::**perturbation_Disp() définie dans //*cycle_base_time_loop.h*, mais surchargée par celle définie dans la dll spécifique :

perturbation_Disp(…);

//sauvegarde des variables choisies par l'utilisateur au temps t+dt dans des fichiers //pour post-processing en appelant la fonction vide **Time_loop::**save() définie dans //*cycle_base_time_loop.h*, mais surchargée par celle définie dans la dll de //sauvegarde spécifique :

save(…);

}

Cette séquence est réalisée tant que "*Iteration <= Max_Iteration && Sim_time <= Duration*".

Détail sur la clôture de l'application

C6) Enfin, la dernière fonction appelée, **Close_system::**Terminate_simulation(…) est déclarée dans le fichier *cycle_base_close.h* et définie dans *cycle_base_close.cpp*.
Trois étapes successives y sont réalisées :

1. Création de l'objet AL de la classe **Array_List** qui est déclarée dans *cycle_base_list_Array_conversion.h* et définie dans cycle_base_list_Array_conversion.cpp. Cet objet réintroduit dans les listes à chaînes liées certaines caractéristiques physiques des éléments discrets qui étaient présentes dans le tableau d'éléments discrets (tableau SDE). Ceci permet de remettre à jour les chaines liées d'éléments discrets avant leur sauvegarde dans les fichiers.
2. Création de l'objet SSIM de la classe **Save_simulation** déclarée dans *cycle_save_simulation.h* et définie dans *cycle_save_simulation.cpp*. Cet objet intègre plusieurs fonctions qui vont sauvegarder les chaines liées d'ED (éléments discrets), **Save_simulation::**Write_list_SDE, le tableau d'ED(…), **Save_simulation::**Write_array_SDE(…), la liste des voisins pour chaque ED, **Save_simulation::**Write_SDE_list_NEGB(…), et enfin la liste de chaîne liée des liaisons entre ED, **Save_simulation::**Write_list_LINK(…).
3. Enfin, création de l'objet SSDE_XYZ de la classe **Save_SDE_xyz**, déclarée dans *cycle_save_sde_xyz.h* et définie dans *cycle_save_sde_xyz.cpp*. Cet objet utilise une

fonction qui écrit sous format ascii, un fichier qui contient les caractéristiques de base des ED. Ceci permet de visualiser rapidement la configuration du système en fin de simulation. Cette étape est toutefois redondante avec la phase de sauvegarde réalisée dans la **dll** de sauvegarde spécifique, mais présente par défaut si rien n'est fait par l'utilisateur au niveau de la dll de sauvegarde.

Détermination des interactions : création des liaisons

Deux classes sont dédiées à la détermination des interactions entre les éléments discrets. La première, **Grid** est déclarée dans le fichier *cycle_links_detection_grid.h* et définie dans le fichier *cycle_links_detection_grid.cpp* et la seconde, **Heap_SRT** est déclarée dans le fichier *cycle_links_detection_heapsort.h* et définie dans le fichier *cycle_links_detection_heapsort.cpp*. L'une ou l'autre méthode est utilisée suivant le choix exprimé dans le fichier paramètres principal. Généralement, la méthode de découpage spatial par volume (définie par **Grid**) sera plus performante lors de distributions compacts avec peu de variation en terme de taille d'éléments discrets, l'autre méthode, utilisant un algorithme de tri (par heapsort) et caractérisé par la classe **Heap_SRT**, est plus indiquée dans le cas où les conditions précédentes ne sont pas réunies. Les deux prochains paragraphes décrivent plus en détails le déroulement de la détermination des interactions.

Méthode de détection des interactions par découpage volumique de l'espace

Il a été vu, dans le paragraphe au point 5 du chapitre sur le **Détail de l'initialisation du système**, que la fonction **Link_setup**::Link_creation qui se trouve dans le fichier *cycle_base_setup_LINK_list.cpp*, présente deux options. L'une est l'utilisation de la méthode de grille définie par la classe **Grid**, l'autre l'utilisation de la méthode de tri définie par la classe **Heap_SRT.** Dans le cas du choix de la méthode de grille (ou découpage spatial par volume), le constructeur DETECT_LINK_BY_GRID = new **Grid** est créé dynamiquement. Ce constructeur appelle la fonction **Grid::**Initialisation_Grid(…), qui initialise la taille de l'espace à discrétiser si celui-ci est imposé au niveau des fichier de paramètres.
Les fonctions qui sont appelées sont soit, **Grid**::LINK_INIT(…) soit **Grid**::LINK_RT_fixe_init(…). La seconde est choisie si l'option inscrite dans le fichier paramètres impose à l'espace à discrétiser de rester invariable et de ne pas alors prendre en compte les interactions des éléments discrets une fois sortis de cet espace fixe. A quelques différences près, ces deux fonctions procèdent à l'initialisation de la discrétisation de l'espace contenant les éléments discrets, en appelant dans l'ordre les fonctions suivantes :

1. **Grid::**Get_max_size(…) : détermination des coordonnées des éléments discrets en position extrême du système.
2. **Grid**::Cell_Setup(…) : initialise le découpage de l'espace et crée un tableau de volumes spatiaux.
3. **Grid**::Store_SDE(…) : suivant la position des éléments discrets, ceux-ci sont rattachés à l'élément "volume spatial" correspondant.

4. **Grid**::Check_for_links(…) : Suivant le critère concernant la distance d'interaction, si celui-ci est rempli, la fonction Links_properties->store_link(…) (c'est à dire, **Links::**store_link(…) qui est définie dans *cycle_links.cpp*) est appelée. Cette fonction va vérifier si l'interaction n'existe pas. Si ce n'est pas le cas, elle appelle alors une autre fonction**, Links::**create_link(…) qui va d'une part, stoker l'interaction dans la liste chaînée list_LINK et d'autre part, appeler soit la fonction lpfnDll_new_link() s'il s'agit de la création d'une liaison, soit lpfnDll_new_contact() s'il s'agit de la création d'un contact. Ces deux dernières fonctions sont elles définies dans la **dll** spécifique d'interaction.

Par la suite, cette méthode est sollicitée lors de l'exécution de la boucle en temps dans, *cycle_base_time_loop.cpp*, quand la fonction check_new_contact(…) est appelée. Soit la fonction LS->DETECT_LINK_BY_GRID->LINK_RT(…) (c'est à dire **Grid::**LINK_RT(…)) est appelée, soit, LS->DETECT_LINK_BY_GRID->LINK_RT_fixe(…) (c'est à dire **Grid::**LINK_RT_fixe_init(…)) si l'espace de détection est imposé fixe. Ces deux fonctions, définies dans *cycle_links_detection_grid.cpp* vont faire des parcours similaires à **Grid**::LINK_INIT(…) ou, **Grid**::LINK_RT_fixe_init(…).

Méthode de détection des interactions par tri

Dans le cas du choix de la méthode par tri, le constructeur DETECT_LINK_BY_HEAPSORT = new **Heap_SRT** est créé dynamiquement dans la fonction **Link_setup**::Link_creation(…), présente dans le fichier d'initialisation des interactions, dans le fichier *cycle_base_setup_LINK_list.cpp*. Ce constructeur appelle la fonction **Heap_SRT**::Initialisation_HeapSort(…), qui initialise la table de tri. Suite à cela, la fonction DETECT_LINK_BY_HEAPSORT->LINK_INIT (c'est à dire la fonction **Heap_SRT**::LINK_INIT(…) définie dans *cycle_links_detection_heapsort.cpp*) initialise les interactions en invoquant la fonction **Heap_SRT**::Check_for_links(…) qui est définie dans *cycle_links_detection_heapsort.cpp*. Cette fonction fait appel à une autre fonction, **Heap_SRT**::test_link(…) présent dans le même fichier, qui après avoir testé la validité du critère de création des interactions, appelle la fonction Links_properties->store_link(…) (c'est à dire, **Links::**store_link(…) qui est définie dans *cycle_links.cpp*) si le cas est favorable. Cette fonction va vérifier si l'interaction n'existe pas. Si ce n'est pas le cas, elle appelle alors une autre fonction**, Links::**create_link(…) qui va d'une part, stoker l'interaction dans la liste chaînée list_LINK et d'autre part, appeler soit la fonction lpfnDll_new_link() s'il s'agit de la création d'une liaison, soit lpfnDll_new_contact() s'il s'agit de la création d'un contact. Ces deux dernières fonctions sont elles définies dans la **dll** spécifique d'interaction.

Par la suite, cette méthode est sollicitée lors de l'exécution de la boucle en temps dans, *cycle_base_time_loop.cpp*, quand la fonction check_new_contact(…) est appelée. Si c'est la méthode de tri qui a été choisie, alors la fonction LS->DETECT_LINK_BY_HEAPSORT->LINK_RT(…) (c'est à dire la fonction **Heap_SRT::**LINK_RT(…), est appelée. Cette fonction un parcours similaire à **Heap_SRT**::LINK_INIT(…).

Destruction des interactions

Pendant le déroulement de la boucle en temps, définie dans *cycle_base_time_loop.cp*p, à l'étape où l'algorithme détermine la valeur des interactions entre éléments discrets, c'est à dire, dans la fonction, **Dynamic_cycle**::Force_determination (définie dans *cycle_base_dynamic.cpp*), un test sur l'état des interactions est réalisé. Pour cela, la fonction Links_properties->fracturation_test(…) (soit la fonction **Links**::fracturation_test(…) définie dans le fichier *cycle_links.cpp*) est appelée. Cette fonction fait appel aux fonctions lpfnDll_test_link(…) et lpfnDll_test_contact(…) suivant la nature de l'interaction. Ces fonctions sont définies dans la **dll** spécifique d'interaction. Si il a été décidé dans ces fonctions que l'interaction entre deux éléments discrets devait être détruite, sa destruction physique s'exécute dans la fonction **Dynamic_cycle::**Force_determination(…).

La dll spécifique de simulation

La **dll** (dynamic linked librarie) spécifique de simulation, est une librairie compilée à part. Cette librairie contient les aspects spécifiques de l'opération de simulation; les conditions aux limites, les sollicitations imposées, etc… Cette librairie est remplie par l'utilisateur puis compilée par celui-ci. Le nom de la dll doit correspondre à celui entré dans le fichier de paramètres (*setup_simulation.sim*) afin qu'elle puisse être appelée.

Cette dll se situe dans le répertoire spécifique au projet traité, exactement dans le répertoire suivant : *C:\myproject\dll\cycle\specific_cycle_dll* si le projet traité s'intitule *myproject* et qu'il est situé sur le volume *C:* .

Sur la figure 8, est représenté sous environnement visual, le projet de la dll de simulation.

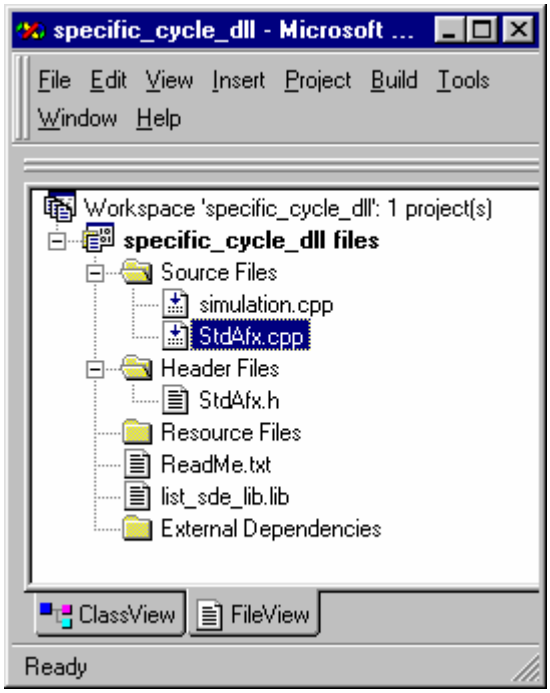


Figure 8. DLL de simulation

Le fichier source, appelé comme désiré par l'utilisateur, ici par exemple *simulation.cpp*, contient les fonctions exportables suivantes :

- read_data_file(…) //lecture du fichier paramètre spécifique
- init_array_tool(…) //initialisation des variables utilisées dans la dll
- perturbation_Force_specific(…) //perturbations ou applications de force dans *cycle_base_time_loop.cpp*,
- perturbation_Disp_specific(…) //perturbations ou applications de déplacements dans *cycle_base_time_loop.cpp*,
- perturbation_velocity_specific(…) //perturbations ou applications de vitesses dans *cycle_base_time_loop.cpp*,

- growth_Deq_adjust_to_radii_specific(…) //grossissement des rayons des éléments discrets si besoin est dans *cycle_base_dynamic.cpp*
- save_data(…) //fonction exportée vers *cycle_base.cpp* et utilisée pour les sauvegardes spécifiques de variables dans *cycle_base_time_loop.cpp*.

Ce sont ces fonctions qui doivent contenir les instructions choisies pour influer sur l'évolution des éléments discrets pendant la simulation.
Noter que la première fonction sert à lire d'éventuels paramètres que l'on souhaite standardiser pour la dll. Ces paramètres sont contenus dans le fichier paramètre intitulé du nom de la dll suivi du suffixe clc.

Généralement, le code généré par l'utilisateur doit se situer entre les bornes suivantes :

```
//+++++++++++++++++++++++++++++++++
// ADD YOUR CODE HERE
//+++++++++++++++++++++++++++++++++
```

et

```
//+++++++++++++++++++++++++++++++++
// END OF YOUR CODE HERE
//+++++++++++++++++++++++++++++++++
```

Tout autre ligne doit rester intacte.

La dll spécifique d'interaction de simulation

La **dll** (dynamic linked librarie) spécifique d'interaction de simulation est elle aussi, une librairie compilée à part. Cette librairie contient les propriétés spécifiques aux interactions ayant lieu entre les éléments discrets durant la simulation. Cette librairie est remplie par

l'utilisateur puis compilée par celui-ci. Le nom de la dll doit correspondre à celui entré dans le fichier de paramètres (*setup_simulation.sim*) pour celui définissant la nature des interactions mis en jeu dans la simulation.

Cette dll se situe dans le répertoire spécifique au projet traité, exactement dans le répertoire suivant : *C:\myproject\dll\cycle\cycle_links_dll* si le projet traité s'intitule *myproject* et qu'il est situé sur le volume *C:*
La présentation de cette librairie sous visual est similaire à celle des autres librairies.

Le fichier source, appelé comme désiré par l'utilisateur, dans tous les cas de figures, contient au moins les fonctions exportables suivantes :

- new_link(…) //nature des interactions de première catégorie
- new_contact(…) // nature des interactions de deuxième catégorie
- test_link(…) //critère de destruction des interactions de première catégorie
- test_contact(…)//critère de destruction des interactions de deuxième catégorie

Par catégorie il est entendu qu'il est possible de faire coexister des interactions de deux natures différentes.
Ces fonctions sont exportées vers *cycle_links.cpp*.

La dll spécifique de sauvegarde de simulation

La **dll** (dynamic linked librarie) spécifique de sauvegarde de simulation, est une librairie compilée à part. Cette librairie définie les informations spécifiques à sauvegarder pour produire des "instantanés" à intervalles réguliers

Le nom de la dll doit correspondre à celui entré dans le fichier de paramètres (*setup_simulation.sim*), additionné de ***save*_snap.dll** afin qu'elle puisse être appelée (exemple : simulation_save_snap.dll).

Cette dll se situe dans le répertoire spécifique au projet traité, exactement dans le répertoire suivant : *C:\myproject\dll\cycle\save_snap_dll* si le projet traité s'intitule *myproject* et qu'il est situé sur le volume *C:* .

Le fichier source, appelé comme désiré par l'utilisateur, dans tous les cas de figures, contient au moins la fonctions exportable suivante :

- fnSave_snap_dll(…) //fonction contenant les informations spécifiques pour créer une sauvegarde de l'ensemble des éléments discrets en vue de faire un "snap" pour un temps t donné. Cette fonction est exportée vers *cycle_base.cpp* et est utilisée pour les sauvegardes spécifiques *cycle_base_time_loop.cpp*.

Librairies statiques

Ces librairies sont définies dans un projet intitulé "source_lib" et elles contiennent certains outils de stockage et de manipulation utilisés dans les différentes étapes de SDEC

Librairie statique définissant la liste "chaîne liée" des éléments discrets

Cette librairie est définie dans le sous-projet, "list_sde_lib" et elle est utilisée dans les trois applications de SDEC, à savoir "*generate.exe*", "*modify.exe*" et "*simulation.exe*".
Deux classes lui sont associées :

- **SDElist**, qui définie les propriétés des éléments discrets.
- **SDEdllist**, qui contrôle l'accès aux objets de la classe **SDElist**.

La librairie générée est, *list_sde_lib.lib*. Elle doit être compilée avec les applications qui l'utilisent.

Librairie statique définissant la liste "chaîne liée" des liaisons

Cette librairie est définie dans le sous-projet, "list_link_lib" et elle est utilisée dans l'application "*simulation.exe*".
Là aussi, deux classes lui sont associées :

- **LINKlist**, qui définie les propriétés des interactions (ou liaisons).
- **LINKdllist**, qui contrôle l'accès aux objets de la classe **LINKlist**.

La librairie générée est, *list_link_lib.lib*. Elle doit être compilée avec les applications qui l'utilisent.

Librairie statique définissant la liste "chaîne liée" des voisins de chaque élément discrets

Cette librairie est définie dans le sous-projet, "list_negb_lib" et elle est utilisée dans l'application "*simulation.exe*".
Là aussi, deux classes lui sont associées :

- **NEGBlist**, qui contient l'identité de l'élément discret voisin,
- **NEGBdllist**, qui contrôle l'accès aux objets de la classe **NEGBlist**.

La librairie générée est, *list_negb_lib.lib*. Elle doit être compilée avec les applications qui l'utilisent.

Librairie statique définissant le tri par empilement

Cette librairie est définie dans le sous projet, "heapsort_lib" et elle contient l'algorithme de tri par empilement. Elle est utilisée dans l'application "*simulation.exe*" pour les recherche de contact (voir le chapitre **Méthode de détection des interactions par tri**).

Cette librairie possède une classe : **BinaryHeap**.

La librairie générée est, *heapsort_lib.lib*. Elle doit être compilée avec les applications qui l'utilisent.

Librairie statique définissant la distribution aléatoire

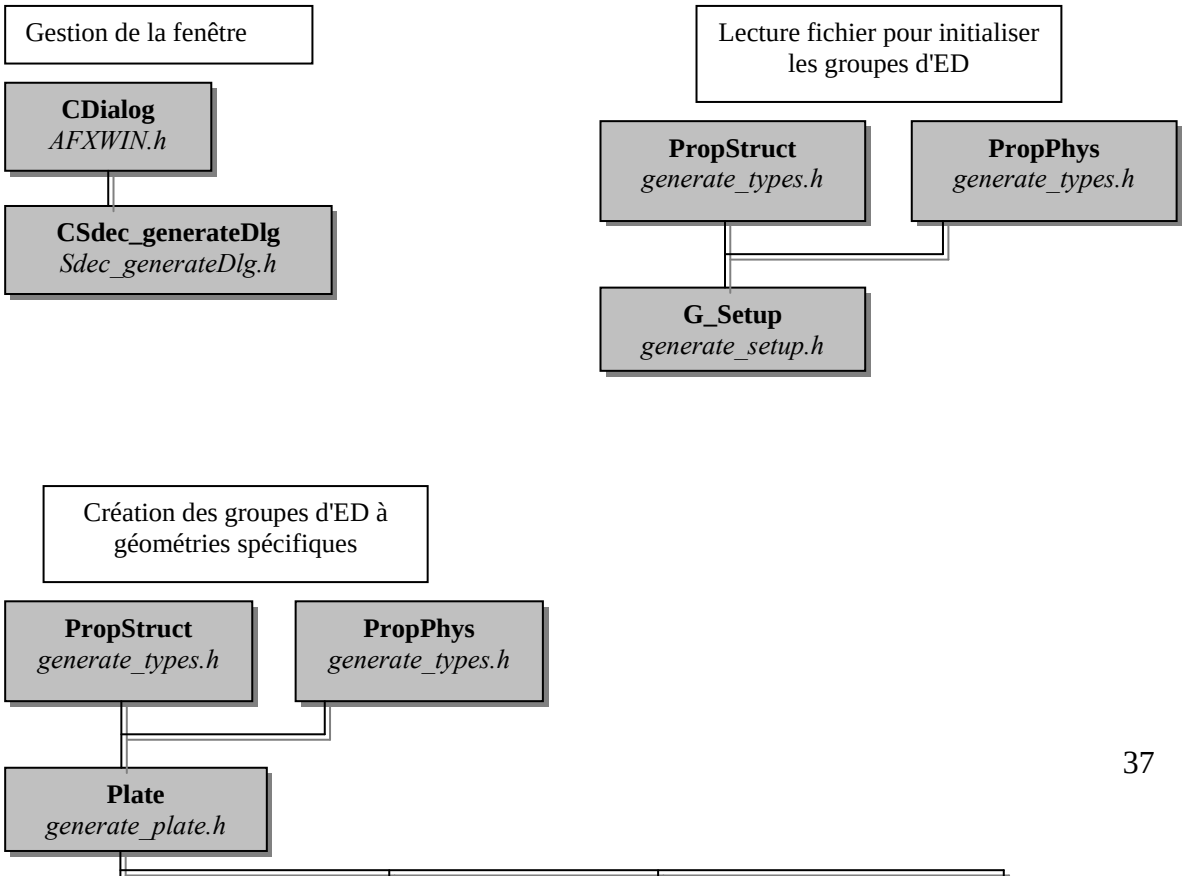
Cette librairie est définie dans le sous projet, "random_lib" et elle contient un algorithme qui produit une bonne distribution aléatoire. Elle peut être utilisée dans toutes les application de SDEC si nécessaire.

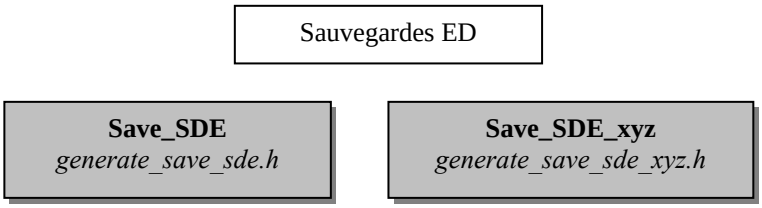
Cette librairie possède une classe : **RandDev.**

La librairie générée est, *random_lib.lib*. Elle doit être compilée avec les applications qui l'utilisent.

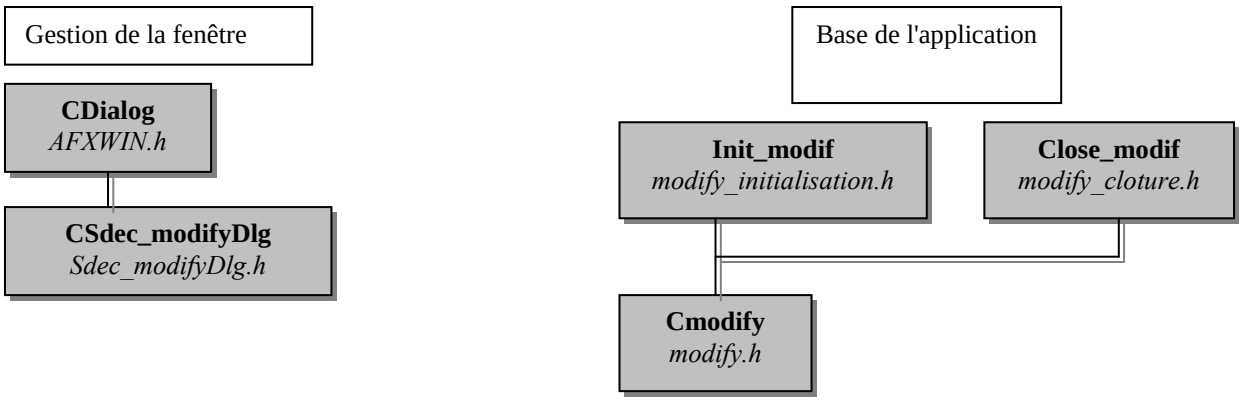
Classes principales dans SDEC

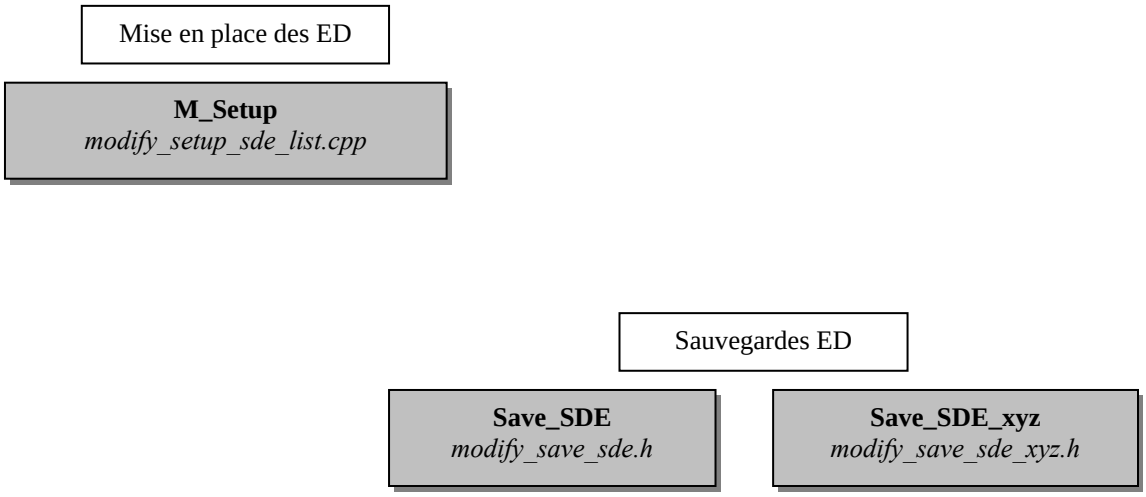
Classes principales dans generate.exe



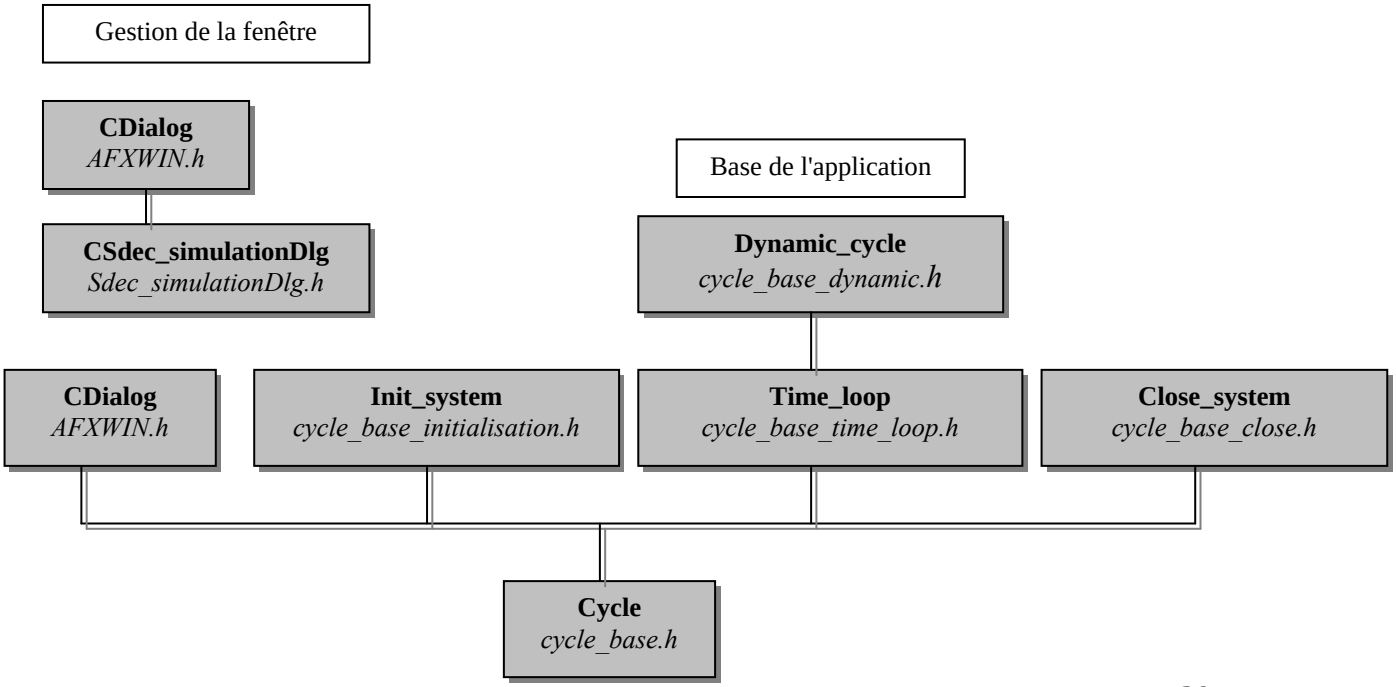


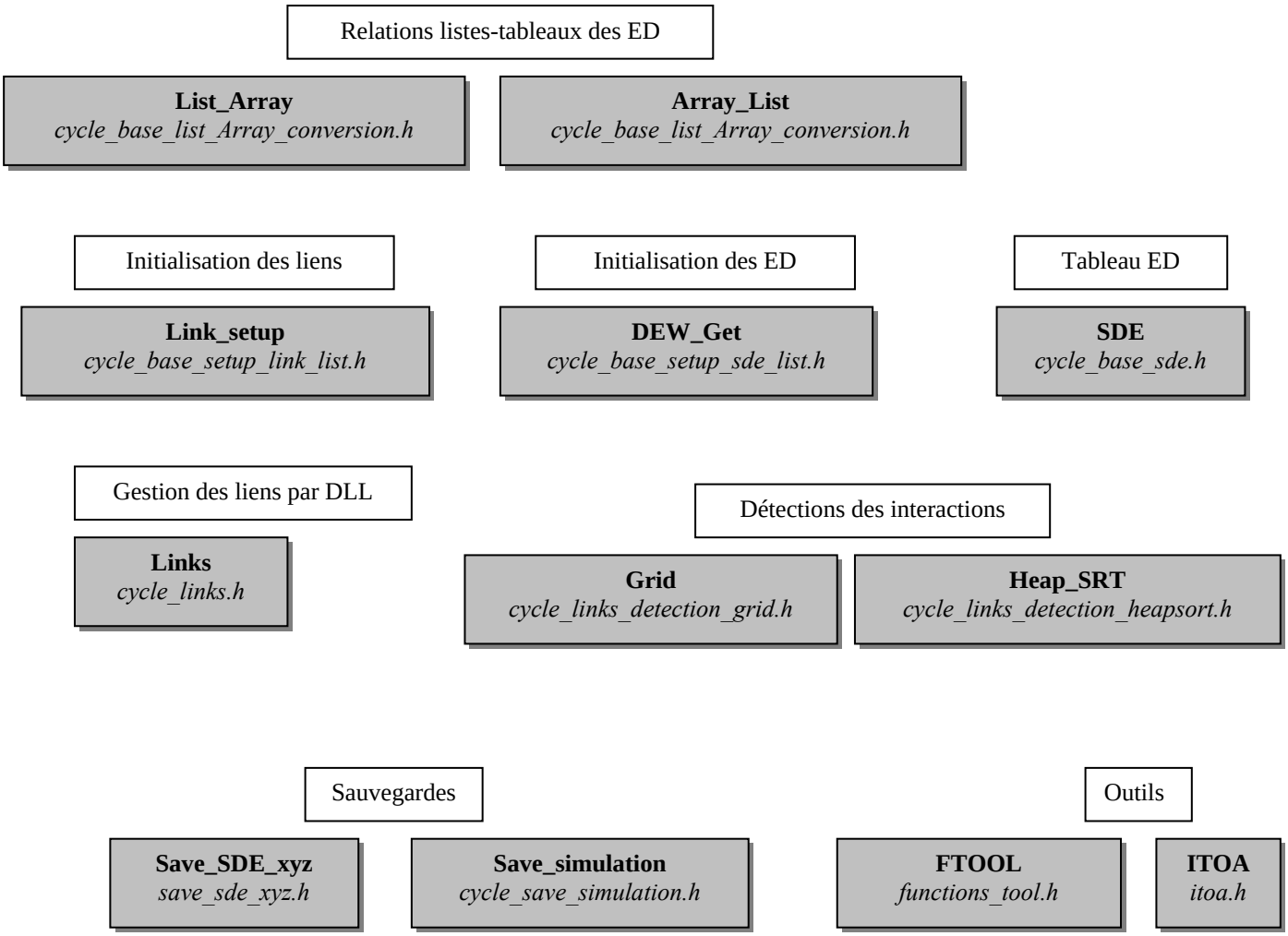
Classes principales dans modify.exe





Classes principales dans simulation.exe




Relations listes-tableaux des ED
List_Array
cycle_base_list_Array_conversion.h
Array_List
cycle_base_list_Array_conversion.h
Initialisation des liens
Link_setup
cycle_base_setup_link_list.h
Initialisation des ED
DEW_Get
cycle_base_setup_sde_list.h
Tableau ED
SDE
cycle_base_sde.h
Gestion des liens par DLL
Links
cycle_links.h
Détections des interactions
Grid
cycle_links_detection_grid.h
Heap_SRT
cycle_links_detection_heapsort.h
Sauvegardes
Save_SDE_xyz
save_sde_xyz.h
Save_simulation
cycle_save_simulation.h
Outils
FTOOL
functions_tool.h
ITOA
itoa.h