# Objectif et modèle conceptuel

## 1. Objectif et périmètre

Le moteur transforme un document source de Modules et de Nodes en un **AST logique indépendant des langages cibles**, puis en informations sémantiques et diagnostics. Une erreur ne détruit ni la source ni la possibilité d'inspecter l'arbre.

Le périmètre livré couvre l'AST, les portées lexicales, les références locales et intermodules, un typage statique strict — primitifs, tableaux, pointeurs, signatures de fonction, génériques avancés (plusieurs paramètres, contraintes exactes et tableaux génériques), classes et lambdas/fermetures — et la sérialisation JSON. S'y ajoutent, sur action explicite, la **simulation locale** déterministe, le **débogage pas à pas** avec points d'arrêt et mémoire pédagogique, les **contrats**, l'**analyse de graphes**, la **simulation graphique simplifiée** et la **transpilation** JavaScript, Python et C expérimental. Une **Discord Rich Presence** optionnelle est fournie par un sidecar C++20 local relié à l'IPC Discord ; elle ne fait pas partie du moteur pur et ne publie que des libellés génériques et des compteurs. Les opérations natives (`Print`, `MathCall`, `ForEach`) restent locales et sans effet système. Moteur JavaScript pur, sans React, DOM, Neutralinojs ni npm ; l'interface y est branchée et déporte les calculs lourds dans un Web Worker avec progression et annulation.

**Exclus :** suggestions contextuelles, exécution automatique, génération de code implicite, analyse de flot avancée, modèles objets et persistance propre du moteur. Le document décrit des fondations, pas un compilateur complet.

## 2. Concepts et relations distinctes

| | |
|---|---|
| **Project** | Document source portant un identifiant, une version de format, une liste de modules et une liste globale de nœuds. |
| **Module** | Unité logique avec identifiant stable, racines syntaxiques, imports et exports explicites. Aucun lien obligatoire avec un fichier physique. |
| **Node** | Enregistrement discriminé par `kind`, propriétaire `moduleId`, propriétés, enfants syntaxiques et références symboliques. |
| **AST** | Arborescence dérivée : les identifiants d'enfants sont remplacés par des constructions hiérarchiques. Les `sourceId` restent stables. |
| **Scope** | Portée lexicale avec parent ; elle ne correspond pas nécessairement au module propriétaire. |
| **Symbol** | Déclaration accessible depuis une portée, ou alias d'import, associé à son nœud et à un type statique résolu ; un échec reste diagnostiqué sans type de secours. |
| **Contract** | Contrainte déclarative optionnelle (`positive`, `nonNegative`, `nonEmpty`) attachée à une variable, une constante ou un paramètre et évaluée pendant l'exécution. |

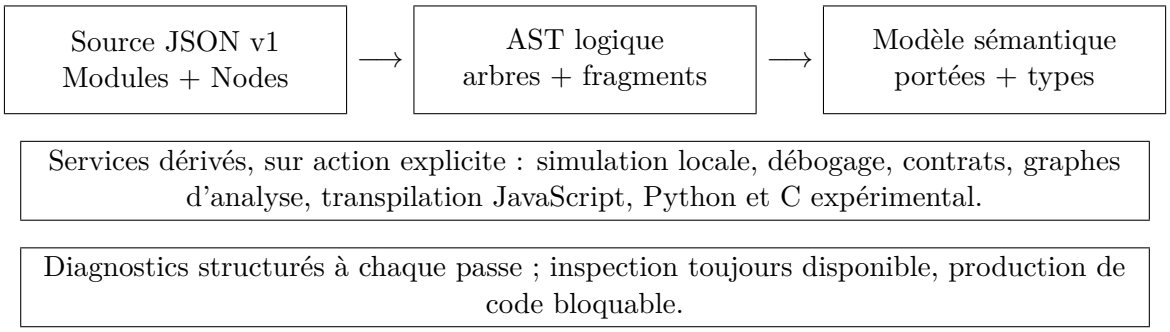
**Trois relations indépendantes :** appartenance par `moduleId`, imbrication par `children`, référence par `references.target`. Une référence vers une déclaration ne la transforme pas en enfant de l'appelant. Une connexion graphique n'a donc aucun sens logique implicite.

L'ancienne catégorie source `main` est retirée. Un projet valide contient zéro ou une seule fonction

portant `properties.entryPoint: true`, indépendamment de son nom ; toute désignation supplémentaire produit `ENTRY_POINT_DUPLICATE` et un marqueur mal placé `ENTRY_POINT_INVALID`, sans supprimer les données invalides. Aucun nom ni signature cible implicite : simulation, débogage et transpilation exigent un lancement explicite. Les sous-modules ne sont pas introduits.

# Architecture et invariants

## 3. Flux et responsabilités




| | |
|---|---|
| `model.js`, `nodeSchema.js` | Contrats JSDoc, fabriques sans génération d'identifiant, catégories, slots et opérateurs centralisés. |
| `ast.js` | Indexation, reconstruction des enfants, contrôle de structure, cycles, parents multiples et propriétaires ; parcours et fragments détachés. |
| `semantics.js` | Collecte des portées et déclarations, contrôle des imports/exports et résolution des références. |
| `typeAnalysis.js`, `primitives.js` | Annotations obligatoires, signatures, types construits et générique T ; dix primitifs, bornes et arrondis. |
| `numericConstants.js` | Contrôles des expressions constantes : conversions, dépassements, sous-flux et division par zéro. |
| `diagnostics.js`, `serialization.js` | Codes stables, gravité, localisation ; lecture et écriture JSON sans migration implicite. |
| `sourceMutation.js`, `nodeFactory.js`, `actionResolver.js`, `contextResolver.js` | Édition contextuelle atomique : actions disponibles, cache d'analyse, insertion, mise à jour, suppression et réordonnancement. |
| `simulation.js`, `contracts.js` | Interpréteur local déterministe, trace, valeurs observées, opérations natives et contrats vérifiés après chaque instruction. |
| `debugging.js`, `reachability.js` | Session de débogage déterministe (états préparés, points d'arrêt, sauts) et analyse conservatrice d'accessibilité. |
| `nodeGraphs.js`, `moduleLinks.js` | Graphes de dépendances et de flot de contrôle d'un sous-arbre ; liens intermodules réellement utilisés. |
| `nodeTransfer.js`, `moduleInterface.js`, `deadCode.js`, `deletionImpact.js` | Transferts de sous-arbres, exports/imports de module, déclarations inutilisées et impact d'une suppression. |

Ces fichiers sont dans `src/utils/logic/` ; l'API publique est `index.js`. `analyzeProgram(source)` renvoie `source`, `ast`, `semantic`, `diagnostics` et `canGenerate`, sans modifier la source. Les backends vivent dans `src/utils/transpilation/` ; JavaScript, Python et C expérimental sont actifs ; C++ et Java indisponibles.

## 4. Structures et invariants

- Identifiants de nœuds uniques dans le projet ; identifiants d'import uniques dans leur module. Aucun identifiant n'est recréé pendant les passes.
- Un enfant syntaxique a au plus un parent et appartient au même module. L'ordre des listes de slots est significatif, pas celui du tableau `nodes`.
- Déclarations : variables, constantes, fonctions, paramètres et alias de types. Instructions : blocs, retours, affectations, conditions, boucles (`break`/`continue` compris), parcours `ForEach`, instructions d'expression et affichage. Expressions : littéraux, références (y compris de fonction), appels, opérations unaires (`&`, `*`) et binaires, tableaux, accès à un élément, opérations natives et conversions.
- Un slot requis absent produit `Hole`. Un lien impossible produit `Invalid`. Les fragments détachés restent visibles ; la source conserve également les enregistrements malformés ou dupliqués.
- Annotations obligatoires des déclarations et littéraux : `primitive` ou alias transparent `named`. Signature `function` dérivée pour les fonctions. Dix primitifs : `short int`, `short uint`, `int`, `uint`, `float`, `double`, `string`, `char`, `boolean`, `void` (retour uniquement). Le type tableau `{kind:'array', elementType}` s'annote uniquement sur une variable, une constante ou un paramètre, avec un élément primitif non `void`. Le type pointeur `{kind:'pointer', pointee}` et le type fonction `{kind:'function', parameters, returnType}` s'annotent sur ces mêmes emplacements ; une fonction générique porte le type `{kind:'generic', name:'T'}` sur son retour et ses paramètres seulement. Aucune promotion ni conversion implicite.

# Exemple et résolution des références

## 5. Deux modules, une référence intermodule

Le scénario exécutable `twoModules()` dans `tests/logic/fixtures.js` crée la constante publique `answer = 42` dans `math`, puis la variable `result` dans `app`.

```
Project { formatVersion: 1, id: "project" }
  Module "math"
    rootNodeIds: ["answer"]
    exports: [{ nodeId: "answer", name: "answer" }]
  Module "app"
    rootNodeIds: ["result"]
    imports: [{ id: "answerImport", moduleId: "math",
                nodeId: "answer", localName: "answer" }]

Node "answer" : ConstantDeclaration, moduleId = "math"
  properties: { name: "answer", type: primitive("int") }
  children.initializer: "fortyTwo"
Node "fortyTwo" : Literal, value = 42, type = primitive("int")

Node "result" : VariableDeclaration, moduleId = "app"
  properties: { name: "result", type: primitive("int") }
  children.initializer: "answerReference"
Node "answerReference" : Reference, moduleId = "app"
  references.target: { kind: "import", importId: "answerImport" }
```

`primitive("int")` est ici une notation abrégée pour `{kind:"primitive", name:"int"}`.

L'AST simplifié est le suivant ; chaque construction conserve son identifiant source :

```
Project
  Module math
    ConstantDeclaration [answer]
      initializer -> Literal [fortyTwo] : int
  Module app
    VariableDeclaration [result]
      initializer -> Reference [answerReference] : int
```

La déclaration `answer` n'est pas recopiée sous `result`. Le modèle sémantique associe la référence au symbole importé et au nœud cible `answer`. Sans export ou import explicite, la référence est rejetée.

**Règles de portée :** une portée par module, fonction, bloc imbriqué et `for`. Paramètres et bloc principal d'une fonction partagent une portée. Les noms sont collectés avant résolution ; un doublon local est une erreur, un masquage dans une portée interne est autorisé. Une référence directe par identifiant ne contourne pas les frontières lexicales ou les imports.

# Vérification et évolutivité

## 6. Validation et résultats constatés

Un diagnostic contient `code`, `severity`, `message` et `location = {projectId, moduleId, nodeId, field}`. La localisation désigne le nœud fautif ou le propriétaire d'un slot manquant. Les éléments inapplicables sont nuls. Les codes ne dépendent pas de la langue du message.

Contrôles structurels : identifiants invalides/dupliqués, enfant absent, mauvais propriétaire, catégories illégales, cycles, parents multiples, fragments détachés et slots incomplets. Contrôles sémantiques : référence inaccessible, import/export invalide, conflit de déclaration, point d'entrée dupliqué ou invalide, retour hors fonction, contrôle de boucle hors boucle, affectation d'une constante.

Typage strict : annotations, initialisation, opérateurs, conditions booléennes, signatures/appels, affectations et retours. Les tableaux (littéral, création, accès) doivent respecter le type d'élément annoncé ; les tableaux imbriqués et un alias nommé comme élément sont refusés. Les expressions non typables restent inspectables sans type publié. Le contrôle de retour accepte les blocs et les deux branches d'un `if`, mais ne prouve pas les retours via les boucles. Entiers signés/non signés de 16/32 bits, flottants IEEE 754 de 32/64 bits ; `char` est un scalaire Unicode. Les conversions `CastExpression` sont numériques explicites, avec troncature ou arrondi et contrôle des bornes.

**Vérifications exécutées :** `pnpm test`, runner natif Node : **770 tests réussis, 0 échec**. Couverture de la construction de l'AST, des portées et références, du typage et des tableaux, des pointeurs et de l'écriture à travers eux, des types fonction et références de fonction, des génériques avancés et des lambdas/fermetures, des bornes et arrondis numériques, de la sérialisation JSON et des migrations de workspace, de la simulation (tableaux, contrats, fonction choisie, `ForEach` et opérations natives compris), du débogage pas à pas (points d'arrêt et mémoire), de l'accessibilité, des graphes de dépendances et de flot, de la transpilation JavaScript, Python et C expérimental, des transferts de sous-arbres, des exports/imports, des sauvegardes de secours, de la détection des déclarations inutilisées, des notifications par mascotte, de l'aimantation des modules et de la vitesse de simulation.

`pnpm build` : frontend compilé avec succès. Ce contrôle protège le frontend existant ; le moteur séparé est vérifié par les tests Node, pas par la compilation du bundle. Le moteur reste sans React, DOM ni API native ; l'interface ne partage avec lui que les identifiants de module et les libellés. Aucune fonction native nouvelle n'a été ajoutée pour ce document.

## 7. Simulation, débogage et transpilation

Sur action explicite, `simulateProgram` exécute le point d'entrée — ou une fonction choisie avec des entrées typées — sans modifier la source. Le résultat reste en mémoire : sorties, trace ordonnée d'identifiants de nœuds, valeur de retour et variables observées. Budget de pas (1 à 100 000) et profondeur d'appel (200) bornent le travail. Tableaux, pointeurs, `ForEach` par recopie et opérations natives (`Print`, `MathCall`) s'exécutent localement ; `*p` relit et écrit l'emplacement visé. Une fonction générique s'exécute sans spécialisation, avec ses types concrets inférés. Une lambda anonyme s'exécute dans un environnement dont le parent est son environnement de création ; une fermeture retournée garde ainsi ses variables après la sortie de la fonction englobante. L'interface déporte simulation, debug, transpilation et vérifications dans un Web Worker avec progression et annulation immédiate.

Les cartes Screen, Gauge, LED, ProgressBar, BarChart, XY plot, Oscilloscope et Heatmap sont adossées à des modules ordinaires, sans nouvelle construction logique. BarChart affiche un `int[]` et exporte `SET_VALUE`, `SET_DATA`, `CLEAR`. XY plot est un traceur 2D interactif propulsé par Mafs : `SET_FUNCTION` reçoit une fonction `(double) -> double` et échantillonne 64 points, `SET_RANGE` fixe les bornes et `CLEAR` efface. L'oscilloscope multicanal est propulsé par uPlot : `ADD_CHANNEL`, `SET_CHANNEL`, `REMOVE_CHANNEL`, `SET_RANGE` et `CLEAR` pilotent jusqu'à six canaux et 256 échantillons. Mafs et uPlot restent côté interface ; le moteur ne dépend ni de React ni du DOM. Heatmap conserve une grille fixe de valeurs entières et exporte `SET_CELL(x,y,value)` et `CLEAR`. Sa couleur d'intensité suit le thème et les pas de Run/Debug, sans lecture d'images à 24 fps. Les vues aperçu/code sont persistées ; ces widgets restent exclus des cibles de transpilation.

Les contrats `positive`, `nonNegative` et `nonEmpty` sont évalués après chaque instruction ; leurs violations sont signalées sans interrompre le programme. Le débogage réutilise le même interpréteur et parcourt des états préparés (suivant, sortie de scope, retour, redémarrage), avec points d'arrêt visuels et sauts qui s'y arrêtent, mémoire pédagogique (code, globales, pile d'appels) et simulation graphique simplifiée des fonctions atteignables et de leurs appels directs, sans persistance ni appels indirects.

La transpilation délègue à des backends enregistrés : JavaScript, Python et un C expérimental sont actifs ; C++ et Java restent indisponibles. Tous consomment le même AST analysé et résolvent les noms, imports et références via le contexte commun ; aucun ne réinterprète la source. Les diagnostics bloquants empêchent toute génération et le code produit n'est jamais exécuté automatiquement.

### 7.1 Périmètre par cible

| **Construction** | **JavaScript** | **Python** | **C expérimental** |
|---|---|---|---|
| Primitifs numériques, booléen | oui | oui | oui |
| `string` | oui | oui | `const char *` UTF-8 |
| `char` | oui | oui | non |
| Tableaux | oui | oui | non |
| Pointeurs `&`/`*` | cellules | cellules | natifs |
| Références de fonction | oui | oui | pointeurs nommés |
| Lambdas et fermetures | oui | oui | sans capture |
| `ForEach` | oui | oui | non |
| Classes, `new`, `this` | oui | oui | non |
| Génériques | fonction ordinaire | fonction ordinaire | non |
| `MathCall` | partiel | partiel | non |
| Conversions explicites | oui | oui | non |
| Contrats | non émis | non émis | non émis |
| Widgets et entrées | refusés | refusés | refusés |

JavaScript et Python émettent les tableaux, références de fonction, lambdas, `ForEach` (copie), classes et opérations natives prises en charge. Les génériques ne créent pas de copie : une seule fonction ordinaire est émise, avec les types concrets inférés. Le C accepte les primitives numériques, les booléens, les chaînes, les fonctions, le contrôle, les pointeurs, les pointeurs de fonction nommés, les lambdas sans capture et `Print` ; il refuse les tableaux, classes, lambdas capturantes et fermetures, `ForEach`, `MathCall`, conversions explicites, `char`, variables sans initialiseur et adresse d'une constante avec `TRANSPILE_C_UNSUPPORTED`. Les contrats restent propres au simulateur et au Run live.

Le registre `mathNatives.js` est le catalogue unique des opérations SP3 ; chaque backend fournit une table de correspondance par identifiant et refuse explicitement une opération absente. Une fonction ordinaire est émise une seule fois et les imports sont résolus vers le nom généré de la fonction exportée, sans inclusion ni import runtime. En C, une référence de fonction nommée devient un pointeur typé par un `typedef` ; les paramètres, variables, retours et appels via une référence nommée sont émis.

Une lambda sans capture est levée en fonction statique et utilisée comme pointeur de fonction ; une lambda qui capture une variable locale ou un paramètre est refusée.

## 8. Évolution sans couplage prématuré

La simulation, le débogage et la transpilation consomment déjà l'AST et les symboles résolus. Les **suggestions contextuelles** au point d'édition et de **nouveaux générateurs** de cibles restent à venir ; chaque générateur devra refuser les diagnostics bloquants. `canGenerate` signifie seulement « aucune erreur détectée par les passes actuelles », et non une garantie d'exécutabilité. Aucun moteur de suggestion n'est implémenté.

Le **Plugin SDK** local est une couche déclarative séparée du modèle source : un manifeste `.sp3plugin` versionné décrit des widgets, des opérations natives typées et des composites de niveau A. Le registre est configuré avant l'analyse, injecté dans les Workers et revalidé avant exécution. Les widgets de plugin restent des cartes de présentation, sans module ni AST ; les opérations natives sont refusées si leur backend manque. Aucun `eval`, `new Function` ni import de code tiers n'est utilisé ; aucun marketplace, backend distant ou permission native n'est implémenté.

Le format source v1 préserve les champs supplémentaires. Une version source inconnue est conservée et signalée. L'enveloppe workspace est en version 3 : les fichiers versions 1 et 2 sont migrés explicitement à la lecture (drapeau générique historique et `screenScale`) avant validation ; une version future est refusée avant tout remplacement. La sérialisation refuse les données non JSON au lieu de les supprimer silencieusement. Les profondeurs d'analyse et de sérialisation sont bornées pour éviter les dépassements de pile.

La **Discord Rich Presence** reste hors du moteur logique : un sidecar C++20 statique est lancé par Neutralinojs, se connecte à l'IPC local de Discord et reçoit des activités déjà construites par l'interface. Aucun OAuth, token, compte ou backend n'est utilisé ; seuls des libellés fixes et des compteurs de modules/nœuds sont transmis. Le sidecar est compilé pour Windows, Linux et macOS lors de la distribution ; CMake et Zig ne sont requis qu'à la construction. Le réglage `sapphire-discord-presence` est activé par défaut et arrête immédiatement la présence lorsqu'il est désactivé.

Le workspace est un dossier choisi : `workspace.sp3`, images liées dans `images/`, trois sauvegardes directes au plus dans `backups/`, export ZIP. Un document neuf non vierge utilise le dossier persistant `sapphire3-recovery/`.

Le **Run live** est le mode par défaut de l'application : un interpréteur coopératif exécute le programme par tranches, sans budget total, et rend la main au pilote à chaque tranche ou à `screenPresent`. Le pilote applique les entrées et les événements `onClick` entre deux frames, met à jour les écrans et widgets, et permet Pause/Stop. Un curseur règle la cadence du live de 1 à 100 sans redémarrer le worker. En live, le curseur de pas ne sélectionne pas les nœuds : le module actif pulse brièvement, sans ouvrir les propriétés. Les boutons d'entrée lient une fonction `(boolean) -> void` : le hold l'appelle une fois à l'appui, le toggle l'appelle immédiatement puis toutes les 500 ms tant qu'il est actif, et le dashboard exécute la même fonction. Le **Run borné** reste disponible pour précalculer une trace finie avec un budget d'étapes ; le debug et la transpilation conservent leur fonctionnement borné. Le live ne fournit ni trace complète, ni contrats, ni retour arrière.

L'**Export Web Dashboard** produit un unique `dashboard.html` autonome : le moteur de simulation réel y est embarqué et exécuté dans un Web Worker, avec un repli synchrone borné. Le dashboard reprend les onze widgets graphiques et d'entrée et les annotations Comment avec formules KaTeX et polices embarquées, dont l'écran se zoome automatiquement pour remplir sa carte en conservant les proportions et en rognant les bords si nécessaire. Il propose **Run live** avec l'interpréteur coopératif

embarqué et Pause/Reprise/Stop, un Run borné, un curseur de vitesse de 1 à 100, un mode Run autonomous qui rejoue la trace en boucle pour les programmes finis et lance le live continu pour les exemples `SP3_LIVE`, la sortie, une mini-carte des modules et classes avec leurs flèches de dépendance et le nœud courant, des panneaux de contrats par module au même niveau que les commentaires, et une bascule clair/sombre. Il reprend la direction Void/Acid de l'application : tokens, panneaux biseautés sans halo décoratif et polices Inter, Space Grotesk et JetBrains Mono embarquées en data URL. La barre de progression permet de revenir à un pas en cliquant ou au clavier ; l'exécution se met alors en pause. Les actions et les modes portent des icônes SVG inline, avec un toast local qui confirme chaque action. Un sélecteur reprend les langues de l'application. Un second sélecteur propose les thèmes prédéfinis et personnalisés de l'application ; le thème choisi dans Settings reste le défaut, le visiteur peut changer librement de thème et de luminosité, et son choix est mémorisé localement. Le pied de page affiche le logo Sapphire 3, la version utilisée, la mention Threadlock Systems et un bouton GitHub sombre vers le dépôt. Chaque thème embarque sa palette sombre et une variante claire dérivée avec contrôle du contraste ; la palette exportée reste une palette de secours portable. La disposition initiale est un bento à deux colonnes : affichages, tracés, indicateurs, sortie, exécution et commentaires ; les panneaux pleine largeur occupent une colonne, les indicateurs carrés une demi-colonne. Les panneaux sont déplaçables, redimensionnables et masquables ; la disposition reste dans le `localStorage` du visiteur, sans modifier le fichier ni le workspace. La configuration est stockée dans la clef racine optionnelle `dashboard` de l'enveloppe version 3, sans migration de version. Aucun serveur, CDN, police distante ou dépendance externe n'est utilisé ; seule l'ouverture volontaire du lien GitHub quitte le fonctionnement hors ligne. Un préflight exécute le vrai moteur avant l'écriture et refuse les programmes incompatibles. Les noms saisis et les sorties du programme ne sont pas traduits. Le choix de langue est mémorisé dans le navigateur ; l'arabe active la direction droite à gauche. Le libellé du nœud courant et les types de la mini-carte suivent la langue du visiteur. Pour les programmes finis, Run autonomous rejoue la même trace sans renouveler les tirages ; pour les projets `SP3_LIVE`, il lance le vrai live continu. Pause et Stop restent disponibles dans les deux cas ; le retour arrière est réservé à la trace bornée. Les panneaux de contrats restent visibles avant toute exécution : nom du module, nœud concerné et règle traduite. Le Run borné fournit les résultats et les violations ; le live ne les calcule pas et ne présente aucun statut de réussite. Le site de présentation expose douze dashboards générés dans `abstract/visual/dashboards.html` ; `pnpm dashboards:build` et la CI les maintiennent alignés après tout changement du dashboard.

La bibliothèque locale compte 65 exemples (8 simples, 33 moyens et 24 difficiles). Spirograph trace progressivement sa courbe en 12 images (64×40) ; Pendulum wave anime huit pendules en 9 images (56×24). Ces images sont précalculées à la construction du projet puis envoyées par `blit`/`present`, sous le budget de 10 000 pas.

**Classes :** un module `kind:'class'` contient seulement attributs, constantes, méthodes et constructeur, avec visibilité `public`/`private`. `NewExpression` crée une instance, `ThisExpression` la désigne dans les méthodes et `MemberExpression` accède aux membres. Le type `{kind:'class', classId}` est annotable sur variables, constantes, paramètres et retours. Les accès privés sont vérifiés ; les méthodes extraites conservent leur instance. Une classe et ses consommateurs partagent les flèches de dépendance des modules. La transpilation des classes est assurée en JavaScript et Python ; le C expérimental les refuse.

**Limites assumées :** pas de sous-modules, héritage, polymorphisme, réexports ni résolution de fichiers ; un tableau porte un élément primitif unique, sans tableau imbriqué ; les génériques acceptent plusieurs paramètres, des tableaux génériques et des contraintes exactes, mais refusent `T*`, `T[][]` et les génériques locaux ; les lambdas ont un corps expression unique, ne sont pas génériques et leurs paramètres ne sont pas adressés par `&`. Les sauvegardes de secours ne tournent que lors d'un enregistrement direct ; la simulation graphique ne trace pas les appels indirects via une variable de type fonction. Les dépassements constants sont diagnostiqués ; les valeurs reçues à l'exécution ne sont pas bornées globalement par l'analyse. La cible C est expérimentale et volontairement

restreinte ; C++ et Java sont indisponibles. Les futurs backends devront appliquer les mêmes opérations contrôlées, sans rebouclage. L'API complète et les conventions de slots sont décrites dans `docs/LOGIC_API.md`.