CAPÍTULO 3

MÉTODOS FORMAIS
EM
ENGENHARIA DA PROGRAMAÇÃO

ÍNDICE

3.1.- Introdução.

Ainda que a história da evolução de uma área científica ou disciplina seja de inegável interesse para a compreensão do seu estado actual, e tal esteja por fazer com rigor e detalhe relativamente à Ciência da Computação, não se pretende realizar nesta secção tal tarefa, mas tão só apresentar em síntese alguns dos marcos da evolução considerados, pelo autor, como importantes para a compreensão das razões do aparecimento dos formalismos e métodos formais (ou rigorosos) de descrição e especificação de programas, bem como dos desideratos que justificam a adopção dos mesmos.

A primeira e bem reconhecida crise da Programação de Computadores surge nos anos 60, resultante de três factores capitais: o aparecimento de uma nova geração de computadores de maior capacidade; o normal incremento da comple-xidade dos problemas a resolver, em resultado de um aumento das expectati-vas; a incapacidade metodológica e tecnológica sentida, à data, para tratar problemas de tal complexidade. É também neste período que os computadores passam definitivamente do âmbito das máquinas de cálculo para o contexto, mais universal, de máquinas de tratamento simbólico e de processamento de dados.

Sentiu-se então a designada “crise lógica dos anos 60” e, em resultado, uma generalizada mobilização da comunidade académica e científica para a resolver. Nos anos 70, a crise ressurge no seu expoente máximo, em resultado da alar-mante constatação de que, pela primeira vez na história da computação, os cus-tos do “software” superam os custos do “hardware”.

Destas crises resultaram os principais alertas para a necessidade premente de, mais rigorosamente, ser compreendida e explicada a Computação em geral, bem como de capacitar a Engenharia da Programação com ferramentas mais adequadas e produtivas, e ainda de bases teóricas e métodos que cada vez mais se assemelhem em termos de produtividade, rigor e fiabilidade, aos que são uti-lizados nas engenharias mais tradicionais.

A comunidade científica da área respondeu, nos anos 60 e 70, com um surpreendente leque de inovadoras e multi-facetadas propostas, à luz das quais a maioria dos desenvolvimentos actuais são ainda realizados.

As propostas mais radicais assentaram na tese de que a origem de toda a obstrução ao normal desenvolvimento da Computação residia basicamente na adopção de um modelo computacional "limitado" por ser sequencial - *o modelo de Von Neumann.* Backus [Backus 78] ficou como um dos “ex-libris” de tal radi-calismo.

Assim, o esforço natural foi no sentido da procura de arquitecturas e de mo-delos computacionais alternativos. Com o objectivo de minorar os problemas de eficiência, arquitecturas paralelas foram desenvolvidas, algumas procurando explorar o super-paralelismo dentro do mesmo modelo computacional, tais co-mo em [Batcher 80] [Darpa 83], outras procurando

arquitecturas ajustadas a modelos computacionais alternativos, como por exemplo máquinas "dataflow" [Gurd e Treleaven 76] e máquinas de "redução" [Darlington e Reeve 81], e ainda os que procuraram alternativas ao nível dos processadores, como processadores do tipo RISC [Patterson e Sequin 82] ou os *transputers* [Walker 85].

Como sabemos hoje em dia, muitas destas tentativas acabaram por não sobreviver ao modelo que procuravam substituir. É até de salientar que a maior parte destas propostas fracassa pelas mesmas razões por que, anteriormente, se haviam assumido como alternativas, ou seja, a incapacidade do "software" exis-tente ou desenvolvido tirar completo proveito das suas potencialidades.

É no entanto inegável que a Engenharia da Programação tem hoje em dia ao seu dispor "hardware" de grandes capacidades computacionais, traduzido em "main-frames" poderosos, "workstations" sofisticadas, computadores pessoais rápidos, redes com razoáveis velocidades de transmissão e sistemas operativos bastante fiáveis e eficientes.

Porém, do ponto de vista do "software" e da Engenharia da Programação, parece ser evidente que a evolução tem sido bastante mais lenta, principalmente ao nível dos métodos e das técnicas.

Como disciplina científica, a Ciência da Computação surge nos anos 60 com o objectivo de desenvolver as bases teóricas que permitam fornecer explicações rigorosas e tratamento matemático para os mais diversos *processos computacio-nais*. Teorias como a das Funções Recursivas, dos Grafos, dos Autómatos, a Ál-gebra e a Lógica, são muitos dos exemplos de conhecimentos matemáticos que passaram a ser usados em Ciência da Computação.

A Ciência da Computação moderna, tal como a conhecemos actualmente, tem aqui a sua génese, preocupada em possuir uma base teórica e formal, mas suficientemente pragmática para se assumir como uma ciência aplicada à reso-lução dos problemas na área da computação. Esta filosofia é sintetizada por Knuth em [Knuth 71] quando afirma:

> *"A minha experiência é a de que teorias são muitas vezes mais estruturadas e mais interessantes quando são baseadas em problemas reais; de alguma maneira, elas são mais excitantes do que algumas vez poderão ser as teorias completamente abstractas."*

A Engenharia da Programação encontra na Ciência da Computação a sua ciência de suporte. A Ciência da Computação tem disponibilizado à Engenharia da Programação, para além das evoluções e resultados próprios desta, enquanto disciplina de engenharia, resultados de grande valor científico. O objectivo é, conforme o ciclo de evolução tecnológica de

Marchetti [Marchetti 81], a passa-gem da fase empírica à fase de teorização visando atingir a automatização.

Em síntese, a credibilidade e evolução da Engenharia da Programação muito tem a ver com o rigor e formalidade dos seus modelos, métodos e técnicas, para além dos princípios específicos que possam ser encontrados.

Resultados foram entretanto sendo conseguidos que vieram trazer algum desenvolvimento à área. No âmbito do generalizado modelo imperativo[1], são de referir os desenvolvimentos relativos ao código, tais como a *programação estrutu-rada* [Dijkstra 76], a *programação modular* [Parnas 72], as linguagens com disci-plina forte de tipos e as linguagens modulares. No âmbito do tratamento intensi-vo de informação (processamento de dados), o *modelo relacional* de bases de da-dos [Codd 70], de base formal, foi decisivo para se atingir o grau de automatiza-ção hoje em dia existente em tal área.

Numa outra linha de evolução, que mais nos interessa aqui considerar, a necessidade sentida por alguns de tratar o código, ou programas fonte, como objectos de carácter matemático, conduziu ao aparecimento de paradigmas e linguagens de programação de base matemática, sendo de salientar as lingua-gens funcionais como Lisp [McCarthy 62], MIRANDA [Turner 85] ou ML [Harper 86], de base lógica como o Prolog [Warren 77] [Colmerauer et al. 79] e de base algébrica como o OBJ [Goguen e Meseguer 82].

Programas escritos nestas linguagens são supostos possuir uma semântica assente em objectos matemáticos bem conhecidos, com propriedades algébricas e lógicas que permitem o seu tratamento matemático, visando a prova matemá-tica da sua *correcção*. Estas linguagens surgem assim não tanto com o objectivo clássico de garantirem eficiência na sua compilação, mas antes procurando que, ao ser-lhes associada uma semântica matemática, tal possa tornar mais fácil o tratamento matemático dos seus programas e, portanto, a própria *verificação da sua correcção* por meio de *prova matemática.*

É neste tipo de preocupações que se pode encontrar a génese do aparecimento dos Métodos Formais de descrição e especificação de programas e siste-mas, ocorrida no início dos anos 80. A *verificação da correcção,* para além de, idealmente, dever ser realizada num contexto matemático, deve ter em conside-ração os próprios requisitos funcionais, entre outros, estabelecidos para o pro-grama. Passa então a ser também fundamental que tais requisitos sejam igualmente captados formalmente, ou seja, sejam eles próprios matematicamen-te representáveis e tratáveis.

Esta ligação entre verificação formal e especificação de requisitos, ou seja, entre métodos de verificação e métodos de especificação, justifica que na secção seguinte se abordem os primeiros, tanto mais que foram os percussores dos segundos, apresentando-se na secção 3.3 os principais

[1] por oposição ao *declarativo* (lógico [Warren 77] ou funcional [McCarthy 62]).

métodos e formalismos de especificação e desenvolvimento de programas e, entre eles, os utilizados nesta tese.

3.2.- Correcção de Programas.

Embora a evolução da Engenharia da Programação tenha demonstrado que propriedades tais como a legibilidade do código, a modularidade, a reutilização, a normalização, a usabilidade e outras, são importantes, é necessário que se faça alguma distinção entre as propriedades que revelam qualidade no *processo* e aquelas que são cruciais quanto à qualidade do *produto*.

Por exemplo, princípios e propriedades tais como legibilidade do código, estruturação, modularidade e reutilização, têm a ver com o processo, ou seja, a sua aplicação pretende garantir a melhor qualidade do processo de fabrico do produto, procurando uma maior facilidade das práticas e uma diminuição dos custos. Porém, as propriedades ligadas à qualidade do produto são diferentes, podendo não ter directamente a ver com a qualidade do processo, ainda que a garantia de tal relacionamento seja um objectivo fundamental de estudo na disciplina, ou seja, procurando garantir que "bons processos" conduzam a "bons produtos".

Duas das cruciais qualidades do produto “software interactivo” são a sua *correcção* e a sua *usabilidade*[2]. A *correcção* é uma propriedade fundamental porque indispensável, ainda que seja uma condição necessária mas não sufici-ente no julgamento final da qualidade de um programa, dado que, como sabe-mos, avaliações de usabilidade realizadas pelos utilizadores podem ser decisó-rias. Relembra-se, a propósito, que esta tese se situa exactamente na conver-gência das duas propriedades tão procuradas.

Sendo uma condição necessária, a correcção dos programas deve portanto merecer, e assim tem acontecido desde os primórdios da computação, uma atenção particular, quer na sua definição substantiva, quer na consequente procura de métodos para a sua rigorosa verificação. A verificação da correcção de um programa ou aplicação, está ligada à verificação de dois requisitos básí-cos: a *terminação* e a *satisfação dos requisitos*. A conjunção destas duas proprie-dades designa-se por *correcção total*, que é o mínimo exigível.

Apesar do desenvolvimento de programas ser uma actividade criativa, rigor é um complemento indispensável, pois só através de abordagens rigorosas se podem produzir produtos mais fiáveis. Porém, rigor é, ainda assim, uma propri-edade intuitiva e não definível de forma "rigorosa". Para além disso, vários níveis de rigor podem ser estabelecidos e atingidos, sendo o mais alto nível de rigor o que se designa por *formalização*.

---

[2] tradução corrente do termo inglês *usability*.

Sendo os programas de computador escritos usando linguagens de programação, torna-se então óbvio que a semântica de um programa está intrinseca-mente ligada à própria semântica das construções da linguagem em que está escrito. Logo, é fundamental que as linguagens de programação sejam dotadas de semântica rigorosa. Por outro lado, programas são constituídos por dados e por entidades que manipulam tais dados, em particular no modelo convencional imperativo. Assim, a semântica de uma linguagem deve dar significado a uns e a outros, o que, ao longo do tempo, nem sempre aconteceu de modo equilibrado. Finalmente, a correcção de um programa está estreitamente ligada à satisfação por parte deste de um conjunto de requisitos, dos quais os requisitos funcionais têm merecido uma particular atenção.

A correcção funcional de um programa apenas pode ser determinada caso o significado funcional deste puder ser confrontado com os requisitos funcionais para o mesmo definidos. Idealmente, o significado do programa e dos requisitos seria estabelecido de modo rigoroso num contexto matemático, desta forma possibilitando uma prova matemática. Porém, por razões que se apresentam a seguir relacionadas com as dificuldades de se estabelecerem significados mate-máticos rigorosos quer para programas quer para especificações, a prática mais corrente vai no sentido da utilização de métodos informais.

Apresentam-se nas secções seguintes os diversos esforços desenvolvidos por forma a que a concepção e o desenvolvimento de programas em particular, e a Engenharia da Programação em geral, possuam uma base de rigor indiscutível e tão necessária. Na apresentação caminha-se da semântica das linguagens, eta-pa básica para que programas possam ter significado, para os métodos formais de concepção e desenvolvimento, que deverão possibilitar o estabelecimento de contextos rigorosos para que os programas possam ser desenvolvidos em con-cordância com as suas especificações, de forma integrada e provadamente cor-recta.

3.2.1.- Semântica das Linguagens.

A possibilidade de se realizarem raciocínios e provas matemáticas sobre a correcção de programas de computador foi uma preocupação imediata dos principais pioneiros da computação (cf. [Goldstine e von Neumann 47], [Turing 49]), posteriormente retomados por outros investigadores da área [McCarthy 63], [van Wijngaarden 66], [Naur 66] e [Floyd 67], ainda que muitos dos resulta-dos obtidos não tenham sido imediatamente relacionados entre si e, consequen-temente, reforçados, por serem resultantes de esforços individuais.

Porém, a *verificação da correcção* dos programas, i.é. a possibilidade de se produzirem argumentos sobre a sua correcção, não pode ser desligada do signi-ficado - *semântica* - das diversas construções sintácticas da linguagem

em que estes são escritos. Daí que igualmente bastante cedo (anos 60), ainda que em paralelo, apareçam referidas na literatura conferências e cursos sobre *descrição formal de linguagens de programação*[3]. Embora a maioria dos esforços na forma-lização de linguagens de programação não tenham tido a ver directa e imediata-mente com preocupações de verificação dos programas mas antes com a cons-trução de compiladores mais eficientes, o resultado observável é uma influência mútua entre as áreas, ainda que mais no sentido da semântica das linguagens para os métodos de verificação do que no sentido contrário, pois poucas foram as linguagens definidas tendo por objectivo facilitar a verificação formal da correcção dos programas nelas escritos, ainda que regras para tal fossem já co-nhecidas [Tennent 77].

Escrito um programa numa qualquer linguagem de programação, a determi-nação matemática do seu significado, indispensável para a posterior avaliação da sua correcção, passa pela definição formal da *sintaxe* e *semântica*, estática ou dinâmica, de tal linguagem de programação.

A *sintaxe*, seja mais abstracta ou mais concreta, descreve a estrutura correcta dos programas numa perspectiva gramatical. A *semântica estática* des-creve restrições estruturais, tais como regras de verificação de tipos[4], em geral não adequadamente descritas pelos usuais formalismos sintácticos ditos inde-pendentes do contexto, estabelecendo pois relações entre sintaxe e semântica. A *semântica* define os efeitos e os resultados das diferentes construções sintácti-cas da linguagem. O significado (*semântica*) está pois intrinsecamente ligado à forma (*sintaxe*), pelo que qualquer método de definição semântica de linguagens se baseia neste facto.

A definição da sintaxe concreta de uma linguagem de programação é feita usando o standardizado formalismo designado por *Backus-Naur Form* ou BNF, empregue pela primeira vez na descrição da linguagem Algol 60 [Naur 60]. A de-finição matemática da semântica de linguagens de programação de tipo impera-tivo, foi sempre um problema encarado como relevante, ainda que, dadas certas características destas linguagens, o processo se tivesse revelado de grande com-plexidade matemática e, portanto, não muito atractivo e pouco empregue.

De facto, as linguagens imperativas possuem, por forma a serem eficientes, construções muito ligadas à arquitectura da máquina, logo pouco abstractas em relação à mesma. Assim, *instruções* correspondendo a directivas são as unida-des básicas de programação, e programas meras sequências de instruções cujo objectivo é actuar sobre o *estado* da máquina subjacente. A sequenciação explí-cita das instruções através da utilização de estruturas de controlo, obrigando a que a complexidade de um problema seja resolvida através do seu fracciona-mento segundo uma ordem temporal

[3] cf. a Conferência IFIP sobre "Formal Language Description Languages", bem como o curso do laboratório de Viena da IBM sobre "Semântica das Linguagens de Programação", ambos em 1965.

[4] *type-checking*.

implícita, é outra característica típica e de complexo tratamento matemático. Finalmente, as linguagens imperativas basei-am-se na noção de *variável* de valor mutável no tempo através de uma instrução de *atribuição,* e na passagem de parâmetros por referência, associando dois identificadores à mesma quantidade. Perdem assim, de forma definitiva, uma propriedade crucial em matemática: *a transparência referencial*[5]. A consequência imediata da não obediência a esta propriedade nas linguagens imperativas é a possibilidade de surgirem nos programas *efeitos laterais*[6] que, para além de pre-judicarem a qualidade intrínseca do mesmo em termos de legibilidade, modula-ridade, etc., são de difícil tratamento matemático.

Esta discrepância entre a linguagem de programação cuja semântica se pretende definir e a metalinguagem matemática de definição, levou muitos autores a tentarem definir novas linguagens que, por serem matematicamente bem fundadas, prescindiram da metalinguagem. São hoje exemplos clássicos o Lisp [McCarthy 62], com base na teoria das funções recursivas e no ?-calculus [Church 41], o Prolog [Colmerauer et al. 79] [Warren 77] baseado nos trabalhos sobre mecanização da lógica realizados por Kowalski [Kowalski 74] e ML [Harper 86], todas pretendendo realmente abstrair da arquitectura real e ser um passo em frente relativamente ao tratamento semântico formal dos programas, para que se possam estabelecer raciocínios matemáticos sobre a sua *função de execu-ção,* e não apenas quanto à sua mais ou menos eficiente compilação.

A tabela 3.1 sintetiza as abordagens actualmente melhor referenciadas para a definição da semântica das linguagens de programação (LP). Estes métodos (ou classes de métodos) são de seguida apresentados de forma resumida. A im-portância da sua referência resulta de muitos deles estarem na base dos méto-dos de verificação da correcção de programas que serão introduzidos na secção seguinte, e ainda porque alguns deles, ou seus derivados, são nesta tese empre-gues ou referidos.

Métodos de Definição da Semântica das LP	*Operacional* *Translacional* *Gramáticas Atributivas* *Denotacional* *Axiomático*

Tab. 3.1 - Semântica das Linguagens

[5] Esta propriedade expressa-se matematicamente pela noção de substitutividade de iguais por iguais.

[6] *side-effects.*

Semântica Operacional.

Segundo a abordagem *operacional*, a definição da semântica de uma linguagem de programação pode ser realizada com base numa *máquina abstracta* e num *autómato de interpretação*. Estabelecida a máquina abstracta[7], a semântica de cada construção da linguagem corresponde à forma como esta é executada em tal máquina, sendo tal correspondência definida pelo interpretador abstracto (o *autómato*).

Quando rotinas de uma linguagem de programação concreta são usadas na escrita de tal interpretador estamos perante uma semântica operacional definida como concreta. Quando, em contrapartida, são usados objectos matemáticos, em particular, por exemplo usando uma notação funcional executável como em [Martins et al. 85], estamos perante uma semântica operacional matemática.

Em qualquer dos casos, a abordagem operacional assume uma base interpretativa (contrariamente à abordagem translacional, que se apresentará mais à frente) tal como é ilustrado na figura 3.1.

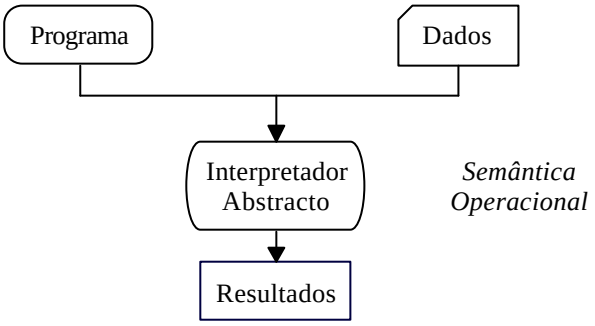

Fig. 3.1 - Semântica Operacional.

É de realçar ainda a semelhança entre esta abordagem para a descrição da semântica de linguagens e as técnicas de prototipagem rápida e de animação de especificações usadas em Engenharia da Programação.

A abordagem operacional tem a sua génese no IBM Vienna Laboratory, aquando da definição da linguagem PL/I, usando a linguagem VDL[8] [Wegner 72]. De salientar que o método de especificação e desenvolvimento de programas designado por VDM[9] [Jones 80], que será apresentado adiante, ainda que tenha sido desenvolvido a partir de VDL, é de base denotacional e não operacional.

Semântica Translacional.

[7] por exemplo, uma *máquina de reescrita* ou uma *máquina de stack.*

[8] *Vienna Definition Language.*

[9] *Vienna Development Method.*

A ideia base da semântica translacional [Stell 66] é exprimir a semântica das construções de uma linguagem usando um esquema de tradução que, para cada programa escrito na linguagem original, realiza a sua tradução para uma outra linguagem, mais simples, mais compreensível, mais tratável em suma.

De notar que a semântica da linguagem se encontra especificada na desi-gnada *função de tradução,* conforme se ilustra na figura 3.2.

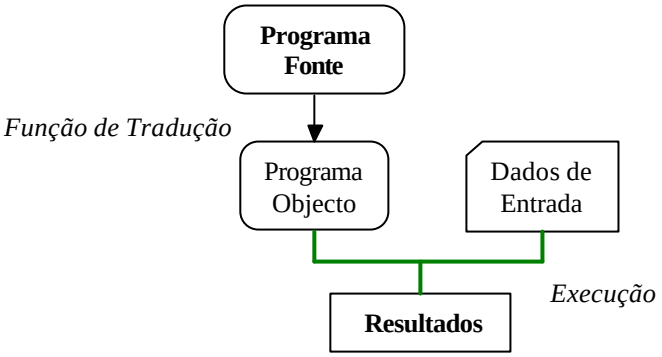

Fig. 3.2 - Semântica Translacional.

Ainda que um tal esquema de tradução possa ser interessante para a futura construção de compiladores da linguagem, dado definir todas as funções de tradução que o compilador deverá implementar caso se use a mesma linguagem objecto, esta abordagem apresenta insuficiências resultantes da sua falta de abstracção, da sua intrínseca dependência da linguagem objecto e, mais impor-tante ainda, por ser recorrente, dada a necessidade de igualmente se prover a linguagem objecto de uma semântica formal. Estas ineficiências da abordagem conduziram ao seu quase completo abandono.

Gramáticas de Atributos.

Especificações gramaticais atributivas adicionam à mera descrição sintáctica de uma linguagem baseada numa gramática, conforme o atrás referido, elementos que visam a descrição da semântica das respectivas construções sintácticas.

As gramáticas de atributos foram introduzidas por Knuth em [Knuth 68] tendo por ideia base um processo designado por *decoração*. Assim, quer a sinta-xe de partida seja concreta ou abstracta, a ideia é decorar as construções sin-tácticas com *atributos* que descrevem as propriedades semânticas dos elementos de tais construções. Quanto às produções da gramática, estas são decoradas com regras que exprimem relações entre os

atributos das construções do lado esquerdo da produção com os atributos das construções do lado direito da mes-ma. A semântica de qualquer entidade sintáctica pode assim ser, em qualquer contexto, vista como o resultado do cálculo dos seus atributos associados.

Duas formas distintas são em geral apontadas como vias para a descrição dos atributos. Uma mais procedimental, ou orientada às rotinas, na qual a ênfase é colocada nas regras, outra mais orientada aos objectos, onde a ênfase é colocada nas construções sintácticas e seus atributos, sugerindo portanto um maior grau de modularidade.

Não possuindo uma reconhecida ligação aos métodos de verificação e de especificação de programas, mas antes a interessantes desenvolvimentos na área específica da semântica das linguagens e dos processadores das mesmas, bem como extensões de grande sucesso na área dos editores estruturados [Reps e Teitelbaum 88], não nos parece de grande utilidade, neste contexto, introduzir mais detalhe sobre esta abordagem, podendo descrições mais completas ser encontradas em [Waite e Goos 84], [Deransart et al. 88] e [Henriques 92].

Semântica Denotacional.

A abordagem *denotacional*, tal como a translacional, baseia-se na definição da semântica de uma linguagem de programação através de um esquema de tradu-ção que associa um significado - *denotação* - a cada construção da linguagem [Scott 70] [Tennent 76] [Stoy 77]. Porém, enquanto que na abordagem transla-cional o resultado da tradução é um programa, nesta abordagem tal resultado é um objecto matemático - a *denotação*[10] da construção -, o que de imediato evita o problema de circularidade atrás referido.

A descrição denotacional de uma linguagem de programação consiste na definição de um conjunto de *funções de significado*[11] que estabelecem as corres-pondências entre as construções sintácticas e os *domínios semânticos*[12], após devida classificação destes, sendo as funções significado indexadas pela classe das construções sintácticas que traduzem. Genericamente, assumem a forma matemática de uma função que estabelece uma correspondência, de um para um, entre cada domínio sintáctico e um domínio semântico, função que é tradi-cionalmente representada por parênteses rectos duplos:

$$??_C : \text{Domínio\_Sintáctico} \rightarrow \text{Domínio\_Semântico}$$

[10] Tradução comum do original *denotation*.

[11] cf. *meaning functions.*

[12] cf. *semantic domains.*

O esquema de tradução é apresentado na figura 3.3.

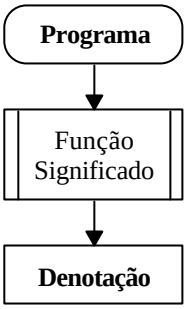

Fig. 3.3 - Semântica Denotacional.

A semântica operacional funcional, ou seja, mais abstracta, e a semântica denotacional possuem, apesar de tudo, grandes similaridades. Em [Meyer 91] a correspondência matemática é mesmo estabelecida, por comparação entre as definições operacional e denotacional da mesma linguagem, e pela constatação de que a versão denotacional usa funções de granularidade inferior às da versão operacional por serem a versão "curried" [Curry et al. 71] das primeiras.

A maior diferença entre as abordagens consiste apenas no facto de que en-quanto na abordagem operacional as construções sintácticas são definidas rela-tivamente a um par <*entrada, estado*>, na abordagem denotacional apenas as construções sintácticas são consideradas, havendo assim um maior nível de abstracção.

Finalmente, note-se que em ambas as abordagens um *modelo* da linguagem é construído, sendo num caso mais operacional e de baixo nível, enquanto que no outro de cariz mais matemático e formal.

Semântica Axiomática.

O método axiomático de definição de linguagens de programação tem a sua génese nos trabalhos de [Floyd 67] e [Hoare 69]. O método axiomático, ao con-trário dos até aqui apresentados, não procura estabelecer qualquer modelo para a linguagem por associação de semântica a cada construção da linguagem, an-tes fornecendo um *cálculo*, com *regras de derivação* e *axiomas*, com base no qual é possível raciocinar sobre propriedades dos programas, estabelecidas usando *predicados* sobre os valores das respectivas variáveis.

Em rigor, porém, o método não fornece directamente uma semântica para a linguagem, não sendo pois um método de descrição semântica. No entanto, o requisito de que a semântica da linguagem satisfaça as regras de derivação e os axiomas do cálculo torna essa semântica num *modelo* e, assim, o cálculo numa descrição exacta da linguagem. O próprio Hoare afirma, no seu artigo

original, que os axiomas e as regras de inferência ou derivação constituem a definitiva especificação da semântica da linguagem.

Floyd em [Floyd 67] usa "diagramas de fluxo de controlo"[13] que decora com fórmulas do cálculo de predicados de 1ª ordem - *asserções* -, relacionando valo-res de variáveis entre si. Floyd sintetiza o problema da verificação da correcção de programas como sendo um problema de prova rigorosa de propriedades enunciadas sob a forma genérica,

> *"Se os valores iniciais das variáveis de um programa satisfazem a relação R1 então, após terminação do programa, tais valores deverão satisfazer a relação R2."*

Hoare, por seu lado, introduz explicitamente *triplos* da forma

{P} S {R}

onde **S** representa uma instrução da linguagem, **P** é um predicado sobre os va-lores que certas variáveis assumem *antes* da execução de **S** (a *pré-condição*) e **R** é um predicado sobre os valores que tais variáveis devem assumir *após* a execu-ção de **S** (a *pós-condição*), com a seguinte interpretação:

> *"Admitindo que **P** é verdadeiro antes da execução de **S**, e admitindo que **S** termina, então **R** é verdadeiro após a execução de **S**"*

Ainda que não constituam uma inovação relativamente às *asserções de Floyd*, os *triplos de Hoare* apresentam no entanto a si associadas regras formais de manipulação, e as bases de uma metodologia para desenvolvimento formal de programas. Hoare apresenta regras de cálculo de consequentes permitindo determinar de forma lógica pré-condições "mais fortes" e pós-condições "mais fracas". Estas fórmulas são conhecidas por *axiomas de Hoare*.

De importância crucial é o facto de o método proposto por Hoare ser, contra-riamente ao de Floyd, *composicional*, i.é., definidas as propriedades das instru-ções simples, as propriedades de uma qualquer instrução composta podem ser deduzidas a partir das suas componentes[14]. Por exemplo, para a instrução com-posta S_1; S_2 (*composição sequencial*) tem-se:

$$\frac{\{P\}\ S_1\ \{R\}\ ,\ \{R\}\ S_2\{Q\}}{\{P\}\ S_1\ ;\ S_2\ \{Q\}}$$

[13] *flowcharts*.

[14] O requisito de composicionalidade nos métodos de verificação é relevante dada a sua extensão aos métodos de especificação.

Para além de apenas ter a possibilidade de verificar correcção parcial, ou seja, não cobrindo a prova da terminação do programa, outro dos identificados grandes problemas do método consiste na dificuldade de se encontrarem as asserções correctas. Floyd já havia identificado este problema, tendo sugerido regra de inferência "para a frente" (i.é. no sentido antecedente - consequente). Segundo esta regra, partindo de uma asserção escrita antes de uma instrução, o problema consiste em encontrar a "mais forte" asserção verdadeira após a exe-cução dessa instrução, ou seja, o *mais forte consequentes verificável*[15].

Mais tarde, o próprio Floyd e outros investigadores (cf. [King 69]) reconhece-ram as dificuldades deste esquema de inferência, pelo que uma regra de infe-rência mais simples, usando inferência "para trás", foi definida e utilizada pelo próprio Hoare no seu trabalho.

Posteriormente, é Dijkstra, em [Dijkstra 74], que sugere um método mais preciso, ao associar à pós-condição não uma qualquer pré-condição mas a, do ponto de vista lógico, *"mais fraca" pré-condicão*[16], escrita como *wp*(S, R), e ao su-gerir regras para tal cálculo, que designou por *transformadores de predicados*.

Por exemplo, para o caso da sequenciação acima referido, a regra seguinte poderia ser considerada

$$wp((S1; S2), R = wp(S1, wp(S2, R))$$

A perspectiva de Dijkstra é bastante diferente da anterior dado que propõe que todas as construções dos programas sejam vistas como transformadores de pós-condições (requisitos) nas "mais fracas" pré-condições que as garantam. Es-ta perspectiva vai de encontro a preocupações de aplicabilidade destes métodos formais na concepção e no desenvolvimento de programas, até aí pouco conside-rada ou impraticável.

É assim que a “programação estruturada” sugerida por Dijkstra em [Dijkstra 76], ao contrário do que muitas vezes se refere, não é um mero marco estilístico mas antes um passo no sentido da aplicabilidade destes métodos a programas imperativos, por eliminação das construções destes que são matematicamente intratáveis. O método de Dijkstra é também mais completo pois permite provar a correcção total.

Como veremos na secção seguinte, onde se aborda o problema mais especí-fico da verificação de programas, o método axiomático, em particular a partir dos trabalhos referidos, conduziu a enormes avanços na área da verificação da correcção de programas, quer sequenciais quer concorrentes.

---

[15] *strongest verifiable consequent*.

[16] *weakest precondition*.

3.2.2.- Verificação de Programas: O código.

A figura 3.4, adaptada de [Berg et al. 82] (e complementada pontualmente pelo autor), apresenta uma taxonomia das mais bem conhecidas abordagens à defi-nição da semântica das linguagens e à verificação de programas.

Das diferentes abordagens à verificação da correcção de programas, são de distinguir, desde logo, duas grandes alternativas: os *testes* e as *provas de cor-recção.* Porque a primeira implica a real execução do programa, designa-se tam-bém por *verificação dinâmica,* enquanto que a segunda, por não implicar execu-ção mas apenas análise documental, se designa por *verificação estática.*

Os *testes* estão directamente ligados a uma das mais antigas definições de *programa correcto,* em particular a que defende que um programa estará correcto desde que a sua execução seja *observacionalmente* correcta, i.é., não mostrando defeitos de execução interna tornados evidentes pela observação dos resultados produzidos ou não (i.é., o seu *output*). Esta perspectiva de *detecção de erros*[17], estreitamente associada aos *resultados observados,* implica, de imediato, que as tentativas de detecção de mau funcionamento sejam apenas aplicáveis numa fase reconhecidamente muito tardia da concepção de aplicações - a *execução*.

A ineficácia do método, directamente associável à sua falta de rigor científico [Dijkstra 76][18], bem como os seus custos, são hoje em dia factos bem reconheci-dos, ainda que a prática não tenha sido substancialmente alterada.

---

[17] *debugging*.

[18] cf. a célebre frase *"... o teste de um programa apenas pode demonstrar a existência de erros, nunca a sua ausência"*.

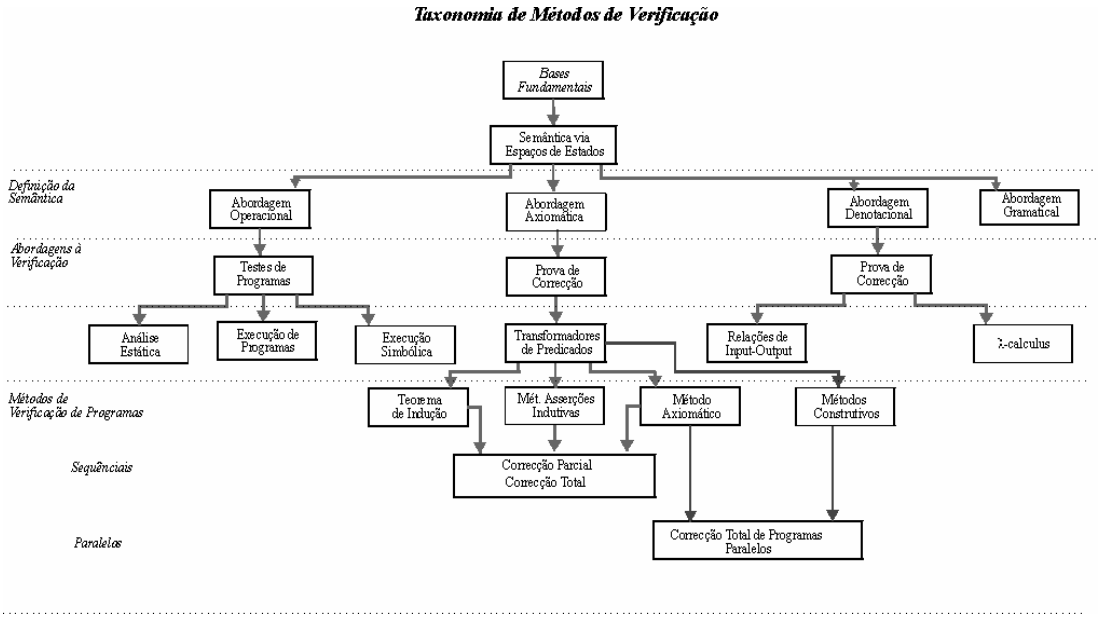

Fig. 3.4 - Taxonomia da Verificação de Programas.

Outras técnicas informais de tipo estático, como por exemplo as *phased ins-pections* [Knight e Myers 93], têm surgido como complementares aos testes, no exemplo em referência apoiada até por uma "ferramenta" em computador.

No entanto, e como salienta Grady num interessante e actual estudo sobre a qualidade do "software" [Grady 93], a boa qualidade do "software" é o resultado da boa qualidade da Engenharia da Programação, sendo a qualidade desta de-pendente do rigor colocado no processo.

A verificação estática e formal da correcção de programas é a alternativa rigorosa apontada. A verificação consiste numa prova matemática de satisfação, onde se procura provar que o programa **P**, uma vez definido matematicamente o seu significado ?**P**?, satisfaz à sua especificação **E**, igualmente estabelecida de modo rigoroso. Ou seja, trata-se de calcular a expressão de prova seguinte, es-crita seguindo a notação apresentada em [Oliveira 94]:

?**P**? *sat* **E**

Como já se referiu, apenas possuindo uma definição rigorosa da semântica da linguagem de programação se torna possível obter o significado de **P.** Por outro lado, só possuindo uma especificação **E** igualmente de significado rigoroso e matemático a prova de satisfação pode ser realizada.

O tipo de suporte matemático empregue em cada uma das abordagens para a definição da semântica das linguagens de programação, e portanto dos pro-gramas, vai decididamente influenciar a definição de ?**P**?, a formulação de **E** e o mecanismo de prova de *sat.*

Por verificação da correcção de um programa, neste contexto formal, deve pois entender-se a prova estática e matemática de que um programa **P**, cujo sig-nificado matemático se denota por ?**P**?, e para o qual se formulou uma especifi-cação precisa **E,** cumpre as condições de correcção, designadamente,

- é finito, isto é, termina;
- satisfaz a sua especificação, ou seja, ?**P**? *sat* **E.**

sendo *sat* a relação lógica de satisfação expressa no formalismo adequado.

A figura 3.4 mostra claramente que a maioria das abordagens para a verifi-cação semântica das linguagens não teve continuação na área da verificação, com a excepção da abordagem axiomática, no seio da qual os mais importantes trabalhos na área da correcção total e parcial de programas sequenciais e con-correntes foram realizados.

No entanto, a maioria destes métodos de verificação de correcção são voca-cionados apenas para a chamada verificação *a posteriori,* ou seja, para a realiza-ção de provas de correcção apenas a jusante do próprio processo de desenvolvi-mento do programa, pelo que não fomentam uma relação estreita entre desen-volvimento e correcção. Em suma, não são métodos *construtivos*.

Os *métodos construtivos* são, do ponto de vista do autor, fundamentais para qualquer método rigoroso de desenvolvimento, atendendo à seguinte ordem de razões:

- *A não ser que preocupações de correcção, especificação e prova, façam parte do próprio processo de desenvolvimento, e aí sejam realizadas em pequenas unidades componíveis, é muito pouco provável que qualquer outro método possa alguma vez ser utilizado. Poder-se-ia até dizer, em sentido figurativo, que o triplo especificação-código-prova necessita de apoio modular;*

- *As provas necessitam de ser realizadas sobre especificações, pelo que também o desenvolvimento do código é realizado tendo por referencial a especificação, e por objectivo o resultado "verdadeiro" para a prova, o que induz igualmente método.*

Os *métodos construtivos* [Dijkstra 76] fornecem as várias condições básicas para que um método possa servir ao desenvolvimento rigoroso de programas, em particular:

- *A possibilidade de se considerar uma especificação **E** como sendo componível, numa estratégia "bottom-up", ou decomponível, numa estra-tégia "top-down", em sub-especificações E_i, segundo o esquema,*

$$E = ? (E_1, \ldots, E_n)$$

em que ? é um operador matemático bem definido.

? *A possibilidade de se considerar um programa **P** como sendo componível, numa estratégia "bottom-up", ou decomponível, numa estratégia "top-down", em sub-programas P_i, segundo o esquema,*

$$P = ? (P_1, \ldots, P_n)$$

onde ? é um construtor sintáctico da linguagem de programação em uso.

? *A possibilidade de se considerarem pequenas provas de correcção, relacionando uma parte do programa, P_i, com uma sub-especificação E_i segundo o esquema,*

$$?\mathbf{P_i}? \text{ sat } \mathbf{E_i}$$

? *A possibilidade de se combinarem as pequenas provas de correcção na prova global de correcção do programa, segundo o esquema,*

$$?\mathbf{P_1}? \text{ sat } \mathbf{E_1} \; ? \; \ldots \; ? \; ?\mathbf{P_n}? \text{ sat } \mathbf{E_n} \; ? \; \; ?? \; (\mathbf{P_1}, \ldots, \mathbf{P_n}) \; ? \text{ sat } ? \; (\mathbf{E_1}, \ldots, \mathbf{E_n})$$

Apesar das muitas dificuldades ainda encontradas na utilização "trivial" de métodos de desenvolvimento de programas baseados na abordagem construtiva, esta perspectiva de desenvolvimento rigoroso de "software" constitui ainda a ba-se da maior parte dos métodos formais de especificação e desenvolvimento de programas apresentados a seguir.

No entanto, uma complementar e inovadora perspectiva de desenvolvimento rigoroso de programas a partir de especificações vem sendo proposta, baseada na ideia de que, com base num *cálculo dedutivo* a estabelecer, de facto *regras de derivação*, poderá ser possível "calcular" classes de programas satisfazendo às especificações dadas [Oliveira 92], cuja aplicabilidade prática e vantagens se procurou salientar, num contexto de desenvolvimento, em [Martins 88a]. Esta abordagem é, de alguma forma, uma versão actualizada das ideias defendidas pela abordagem iniciada por Burstall e Darlington [Burstall e Darlington 77] designada por *abordagem transformacional*.

A ideia fundamental consiste em definir um cálculo de base matemática que garanta que, partindo de uma especificação, por transformações sucessivas seguindo as regras do cálculo, se obtém o código final correcto. Sendo o cálculo semanticamente rigoroso, a correcção estará automaticamente garantida, sendo as provas de correcção dispensadas dada a correcção implícita do cálculo. Este cálculo de programas, designado por SETS [Oliveira 90, 92], serve de base a um esquema de refinamento de

especificações[19] em programas, que se designou por *refinamento transformacional* ou *cálculo de programas* [Morgan e Gardiner 90]. O próprio "cálculo de Oxford" [Morgan e Gardiner 90] pode ser visto como uma no-va ordem de ideias para o cálculo de "weakest preconditions", introduzindo a noção de "invariante de abstracção". Os cálculos de Oxford e SETS possuem al-guma relação na medida em que o que o primeiro assume como heurística a de-terminação dos "invariante de abstracção" que o segundo calcula.

A tabela seguinte sintetiza as diferentes classes de abordagens à verificação da correcção de programas nesta secção referidas.

Determinação da Correcção de Programas	*Testes*	
	Inspecções	
	Verificação	*"a posteriori"*
		Construtiva
	Cálculo	

Tab. 3.2 - Validação de Programas.

3.2.3.- Tipos Abstractos de Dados.

Desde os anos 70 que, aquando do aparecimento das primeiras linguagens com disciplina forte de tipos, a noção de "tipo de dados" foi associada à noção mate-mática de *conjunto de valores*, em aceitável sintonia com a corrente então predo-minante que privilegiava a separação entre dados e instruções. Assim, as opera-ções passíveis de serem realizadas sobre os *valores* de um tipo de dados não fa-ziam parte da sua definição.

Morris em [Morris 73] considera que o modelo adequado para representar matematicamente um "tipo de dados" é uma *álgebra concreta* e não um *conjunto*, e que, portanto, ao conjunto de valores deve ser associado o conjunto de opera-dores que os manipulam.

Esta ideia teórica era suportada na prática pelo que havia já sido consegui-do no mecanismo *class*[20] da linguagem Simula 67 [Dahl 68], bem como pelos esforços de Parnas [Parnas 72] no sentido de que a modularidade dos progra-mas fosse baseada em unidades de implementação, suportadas pelas lingua-gens de programação, promovendo o isolamento das

[19] baseadas em modelos matemáticos.

[20] Na génese da tecnologia orientada aos *objectos*.

representações em conjun-to com as respectivas operações de acesso (*encapsulamento*), bem como a sua opacidade exterior, ou seja, a impossibilidade do utilizador da unidade ter acesso à representação interna dos dados ("*information hidding*"). Tal acesso seria apenas permitido através de uma interface bem definida, constituída pelo conjunto das operações implementadas que explicitamente fossem tornadas pú-blicas pelo implementador do módulo, quaisquer que fossem os mecanismos encontrados pelas linguagens para tal. Estas unidades de encapsulamento com tais características de acesso são genericamente designadas por mecanismos de *abstracção de dados*[21], por analogia com os já existentes mecanismos de *abstracção de controlo* (cf. procedimentos e funções).

As soluções encontradas pelas diversas linguagens de programação para tal tipo de requisitos foram diversas, cabendo referir aqui, sem referências, as solu-ções que foram encontradas nas linguagens de programação mais popularizadas tais como UCSD-Pascal (*units*), Modula-2 (*modules*), CLU (*clusters*) e Ada (*pack-ages*), enquanto imperativas, ML, Hope e Miranda (*abstract type*) enquanto fun-cionais, e *class* em toda a família de linguagens de objectos tais como ABCL/1, Smalltalk, Eiffel, C++ ou Objective-C.

A abstracção comum a todas estas abordagens à implementação de mecanismos de "abstracção de dados", mais ou menos conseguidas[22], é, no entanto, o conceito formal de "tipo abstracto de dados"[23](TAD), introduzido, num contex-to algébrico, por Zilles [Zilles 74], pelo grupo ADJ [Goguen et al. 78] e por Guttag e Horning [Guttag e Horning 78].

Um "tipo abstracto de dados" é uma caracterização abstracta, definicional, das propriedades de um determinado "tipo de dados", sendo tal caracterização abstracta no sentido de que é independente, i.é., afastada, da efectiva imple-mentação do "tipo de dados". Em geral, um novo "tipo de dados" pode ser espe-cificado com base num "tipo abstracto de dados", podendo ser este definido algebricamente ou usando modelos matemáticos. Estas duas possibilidades encontradas irão, de facto, conduzir às duas mais proeminentes abordagens à especificação de programas, designadamente, a abordagem algébrico-funcional e a abordagem por modelos ou construtiva.

A possibilidade de se usarem ambas as abordagens, procurando facilitar a implementação de TAD usando prototipagem, e os cuidados a ter na sua imple-mentação modular usando "tipos opacos" em módulos Modula-2 é apresentada em [Martins 90].

3.3.- Métodos Formais.

---

[21] *data abstractions.*

[22] haveria que distinguir, dentre os mecanismos que possibilitam a criação de tipos definidos pelo utilizador, os que se baseiam em mera "exportação" e os que são verdadeiros construtores de tipos.

[23] *abstract data type* (*ADT*).

3.3.1. - Métodos Formais: Motivação.

O "software" é, indiscutivelmente, um produto singular. Não é feito a partir de materiais em bruto antes sendo um derivado do próprio *processo* subjacente à sua concepção e realização. Assim, sendo o "software" um produto de qualidade ligada quase exclusivamente ao *processo*, dado não ser construído com base em matérias-primas, é preocupante que os processos ainda hoje empregues no desenvolvimento de programas primem pela quase total ausência de rigor e for-malidade.

Não é portanto de estranhar que muita da ênfase colocada no projecto de *software* se situe exactamente no *processo*, suas fases, seus custos normais, seus erros, custos dos erros, rigor, etc. Tratando-se de um problema intrínseco ao desenvolvimento e à qualidade do produto final, a aplicação de métodos formais na captura de requisitos e no desenvolvimento é uma decisão metodo-lógica inteiramente do foro e do interesse da equipa de projecto. O utilizador "vê" o produto final, mede a sua qualidade geral, mas não se apercebe nem discute o processo segundo o qual o mesmo foi construído.

Assim, a adopção de métodos formais é uma decisão de *projecto* e de *processo*, de interesse apenas directamente mensurável pelas equipas de concepção e desenvolvimento de programas. Embora possamos ser, ainda hoje, confrontados com algumas resistências à sua aplicação, em geral por discutíveis razões de índole pragmática[24], a realidade vem demonstrando uma crescente aderência, nas mais diversas áreas, a estes métodos rigorosos de concepção e desenvolvimento de projectos. Como se procurou salientar na secção anterior, a questão fundamental é a introdução de *rigor* no processo.

A descrição formal de um sistema consiste na expressão rigorosa, de base matemática, de determinadas características fundamentais dos sistemas "soft- ware". Podemos descrever formalmente as suas características estruturais, de funcionalidade, de comportamento interno, ou de comportamento externo, ou seja de interacção com outros sistemas. A utilização de formalismos ou notações rigorosas que garantam não ambiguidade, abstracção e até possibilidade de raciocínio sobre determinadas propriedades, é uma condição indispensável.

A não ambiguidade garante interpretação única. A abstracção garante elimi-nação de detalhe e concentração sobre *o que* o sistema deve fazer, e afastamen-to, ainda que temporário, relativamente a *como* o deve fazer. A possibilidade de raciocínio sobre propriedades do sistema deve ser garantida pela tratabilidade matemática do formalismo empregue. Entre o *o quê* e o *como* é igualmente im- portante que se possa considerar o *porquê* rigoroso, ou

[24] em geral consubstanciadas na frase *"Os métodos formais são bons mas são demasiado onerosos."*

seja, a justificação mate-mática dos passos entre as duas etapas anteriores, cf. a figura 3.5.

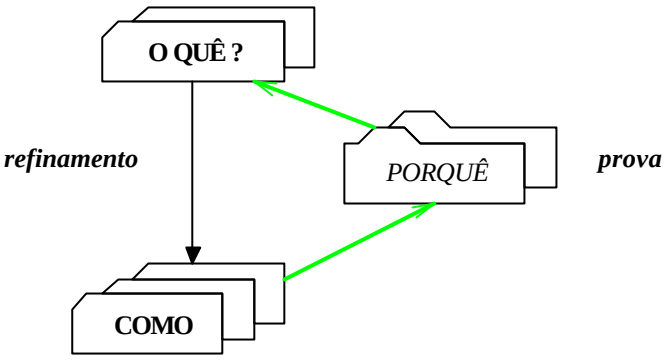

Fig. 3.5 - O Método Rigoroso.

Holt, em [Holt 86], considera que a especificação formal de programas é a técnica dos três Cs: *clareza*, *coerência* e *completude*. Adicionalmente, e ainda segundo Holt, a especificação formal de programas é uma idealização, mas, enquanto tal, importante, pois encoraja cuidadosos e proficientes raciocínios, mesmo que os passos matemáticos dos métodos não sejam completamente cum-pridos.

Estas considerações são muito interessantes pois condizem com a perspec-tiva pessoal do autor sobre a forma de aplicação concreta destes métodos, anga-riada com base em experiência concreta ao longo dos anos.

Se, adicionalmente, os formalismos empregues forem executáveis, então poderemos ter modelos executáveis, i.é. *protótipos* dos sistemas, desde as fases mais preliminares do desenvolvimento, permitindo que desde as fases iniciais, as concepções possam ser operacionalmente testadas e iteradas. Nestes casos, para além do rigor matemático das descrições realizadas, a possibilidade de se utilizar a técnica de *prototipagem rápida*, tida por alguns [Hartson e Smith 87] como meio ideal de diálogo com o cliente (ou o utilizador) para afinação de requisitos, é uma vantagem que em muito pode influenciar na escolha do forma-lismo a empregar.

As principais características dos principais métodos formais de especifica-ção são a seguir apresentadas.

3.4.- Métodos Formais de Especificação.

Ainda que os métodos formais de especificação venham sendo aplicados em múltiplos contextos, como por exemplo a especificação de VLSI, de protocolos de redes, de interfaces com o utilizador, ou de certos componentes de aplicações multi-média, os métodos formais que nos interessa aqui

considerar são os actualmente disponíveis para a especificação dos *requisitos funcionais* de sistemas "software". Estes métodos têm vindo a ser classificados com grau de rigor e amplitude diferente conforme os autores e a sua perspectiva [Cohen et al. 86] [Davis 88], sendo certo que é quase impossível encontrar uma classificação suficientemente disjunta para que seja absolutamente clara.

No entanto, duas perspectivas parecem ser indispensáveis, designadamente a que privilegia a *funcionalidade interna,* ou comportamento interno, e a que tem por visão predominante a *funcionalidade externa,* ou seja, o comportamento que pode ser externamente observável. Em ambas as perspectivas, uma visão se-quencial ou concorrente de tais comportamentos pode ser tomada, ainda que tal influa no formalismo a empregar em cada caso.

Distinguem-se em geral, em termos de abordagem, seis grandes classes de métodos que a seguir se descrevem quanto às suas propriedades e capacidades. Para que um contexto comparativo possa ser estabelecido, estas classes de métodos são apresentadas e analisadas quanto à sua capacidade de expressão de funcionalidade, tempo, e erros ou falhas, quanto à forma como permitem ou não refinamento e ainda quanto ao factor subjectivo da sua clareza.

3.4.1.- Métodos Algébricos.

Os métodos algébricos estão em estreita ligação com a noção de "tipo abstracto de dados" e com a necessidade de se realizar uma caracterização matemática destes de forma completamente abstracta, ou seja, definindo um conjunto de propriedades a respeitar pelas suas operações, e sem fazer qualquer referência ao estado interno.

Formalmente, cada "tipo abstracto de dados" é, neste método, uma unidade básica de especificação, sendo estas especificações vistas como *teorias*[25], ou seja, como pares de *assinaturas* e *axiomas*. *Assinaturas* definem a componente sintáctica do TAD, ou seja, os tipos nele envolvidos (em rigor *espécies*[26]) e a funcionalidade dos operadores. *Axiomas* definem, num dado contexto lógico de interpretação, relações entre operações.

Especifica-se na caixa de texto abaixo a teoria de um TAD *Queue*, parametri-zada pela teoria *Elem* que define a espécie e operações que podem ser feitas com tais elementos. A teoria *Queue* é parametrizada relativamente a uma qualquer

teoria *Elem* que obedeça à assinatura *AssElem*, ou seja, que tenha espécies e operações tais como definidas em *AssElem*, e importa, conforme a cláusula *uses*, as teorias dos naturais Nat e dos booleanos Boolean. As espécies

[25] ver definições no Apêndice B.

[26] *sorts*.

importa-das (cf. *nat* importada de Nat e *bool* importada de Bool) designam-se *espécies primitivas*. Nas espécies *não-primitivas* é vulgar identificar uma espécie designa-da como *principal*, que é aquela que representa os objectos mais relevantes que o tipo exporta (cf. *queue* no exemplo em curso) [Broy et al. 84].

A semântica de cada uma das operações (representadas como *funções*) defi-nidas para uma *queue* é, neste caso, dada pelo conjunto de axiomas da teoria, que definem relações entre elas, interpretadas como sendo propriedades relati-vas ao seu comportamento observável. Daí que muitas vezes se designem os mé-todos algébricos como métodos *orientados às propriedades*.

```
ADT Queue[Elem :: AssElem]
uses Nat, Boolean
sorts queue, elem from Elem, nat from Nat, bool from Boolean
opers
    initq: ? queue                          /* inicialização */
    enqueue : queue x elem ? queue              /* inserção */
    dequeue : queue ? queue                 /* retira primeiro */
    first : queue ? elem                    /* determina primeiro */
    len : queue ? nat                           /* comprimento da
fila */
    empty? : queue ? bool                   /* está vazia ? */
axioms
? q ? queue, e ? elem :
    first(initq) ? ?
    dequeue(initq) ? ?
    len(initq) ? 0
    empty?(initq) ? true
    first(enqueue(q, e)) ?   empty?(q) ? { e          ?
                          ? empty?(q) ? { first(q)
    empty?(enqueue(q, e)) ? false
    len(enqueue(q, e)) ? 1 + len(q)
    len(dequeue(q)) ? len(q) - 1
end.
```

O facto de as operações do TAD serem associadas a funções matemáticas gera por vezes alguma incompreensão, resultante, na maioria dos casos, de muitos anos de experiência e utilização de modelos imperativos. No exemplo anterior, a operação *dequeue*, que remove o primeiro elemento da fila, por ser uma função matemática, não dá como resultado o *elemento* retirado e uma

nova *queue*, como seria de esperar num contexto de *comandos*[27] e não de *funções*[28], mas apenas a nova *queue*, reservando-se a operação *first* para a determinação de qual o primeiro elemento da fila. A composição funcional poderá garantir a funcionalidade de um comando, porém segundo regras bem conhecidas e com significado matemático.

Dentre as operações de um dado TAD, três distintas classes podem sempre ser reconhecidas. Os *construtores* (ou *geradores*) que são o conjunto mínimo de operações necessárias para representar em abstracto (i.é. por *expressões* ou *ter-mos*) todos os possíveis elementos do tipo em definição. Por exemplo, no caso do TAD *Queue*, todos os possíveis valores de uma *queue* podem ser modelados por expressões que envolvem sucessivas aplicações da operação *enqueue* ao valor inicial representado por *initq*. Por exemplo em Queue[Nat], i.é., filas de naturais, teríamos representações de *queues* sob a forma,

"enqueue(initq, 10)"

"enqueue(enqueue(initq, 10), 15)"

"enqueue(enqueue(enqueue(initq, 10), 15), 27)"

Os *interrogadores* (ou *observadores*) são os operadores definidos cujo resultado é de uma espécie primitiva. Assim, estes operadores permitem especificar a semântica dos *construtores* (e também dos *modificadores*) por observação em ál-gebras mais primitivas.

No exemplo apresentado, as operações *first*, *empty?* e *len* são exemplos de *interrogadores*, pois não dão como resultado qualquer representação de um objecto da espécie *queue*, mas antes de objectos de espécies primitivas, sejam elas espécies parâmetro ou não. Finalmente, surgem os operadores semantica-mente mais importantes, os *modificadores*. Designam-se por *modificadores* todos os operadores que se aplicam a termos da espécie principal e dão como resulta-do outros termos da espécie principal, correspondendo assim, simbolicamente, a uma transição de estado do objecto de interesse assim representado. No exem-plo, os operadores *dequeue* e *enqueue* são os únicos modificadores, dado que são os únicos que aplicados a um termo da espécie de interesse, dão como resultado um outro termo dessa espécie, naturalmente alterado.

Note-se ainda que este tipo de caracterização das operações, ou funções, vai ser partilhada com a abordagem por *modelos* (ou *estados explícitos*), onde o problema que se coloca deixa de ser a manipulação de *termos da espécie principal* mas sim a alteração do valor corrente de *um estado representado por um mo-delo matemático*. Poder-se-á agora compreender que a diferença das

---

[27] onde *efeitos laterais* são permitidos e largamente usados.

[28] onde não existem *efeitos laterais* e apenas um resultado é produzido.

abordagens se situa na forma mais ou menos explícita de representação de tal estado, ou seja, *termos* versus *valores de modelos matemáticos*.

Do ponto de vista semântico, a mais usual interpretação dos axiomas é a designada *interpretação equacional* (ainda que não seja a única), i.é., a que estabelece uma relação de *igualdade* entre termos. Estas especificações têm, por tal motivo, a designação de *especificações algébricas equacionais*. Dois tipos de "fecho matemático"[29] para tal conjunto de equações são, em geral, admitidos: "loose" e "initial". As especificações equacionais de tipo "loose" são aquelas que "fecham" o sistema de equações apenas através das regras de inferência equa-cional. As especificações do tipo "initial" usam, para além destas, regras de in-dução estrutural e a regra que determina que se uma igualdade não pode ser derivável das regras de inferência equacional e das regras de indução estrutural, então a mesma corresponde a uma *desigualdade*, ou seja, advogando que, à partida, todos os termos são diferentes.

Axiomas podem assim ser vistos como equações e regras de reescrita, com os quais não só se descreve o comportamento dos operadores sobre os objectos especificados pelos termos que em tal teoria podem ser construídos, como tam-bém possíveis equivalências (na verdade congruências) entre termos. Axiomas podem também especificar termos que representam comportamento de erro.

Por exemplo, o primeiro axioma da teoria apresentada, i.é., first(initq) ? ?, especifica, em abstracto, um comportamento inválido, que é o que resulta de se tentar determinar qual o primeiro elemento de uma "fila de espera" que acabou de ser inicializada (criada) e que, logicamente, está vazia. O axioma determina como *indefinido*, *indefinível* ou *erro*, o resultado de tal sucessão de operações[30].

Em geral, a semântica completa de uma operação só pode ser compreendida analisando todos os axiomas em que a mesma aparece. Por exemplo, relativa-mente à operação de inicialização de uma "fila de espera", que acima se consi-derou criar uma "fila de espera" vazia, tal semântica apenas é garantida pela consideração dos seguintes axiomas:

```
len(initq) ? 0              /* tem comprimento zero */
empty?(initq) ? true            /* não tem elementos */
```

[29] *closure*.

[30] O símbolo ? utilizado é uma simplicação do verdadeiro valor de erro que deveria ser associado à espécie do termo. Assim, a cada espécie um termo de erro deveria ser associado. A complexidade da semântica de erro, de desnecessária introdução aqui, é tratada, por exemplo, em [Ehrig e Mahr 85].

Em alguns casos, axiomas condicionais devem ser considerados, já que o significado da operação pode ser dependente do contexto em que a mesma é executada. No exemplo, o axioma condicional

first(enqueue(q, e)) ? empty?(q) ? { e ?
? empty?(q) ? { first(q)

indica duas possíveis igualdades, ou seja, duas possíveis interpretações do ter-mo *first(enqueue(q, e))*, a primeira num contexto em que a "fila de espera" se en-contra vazia, situação particular em que o elemento inserido passa a coincidir com o primeiro da fila, e a segunda onde, por não se encontrar a fila vazia, a propriedade apresentada apenas garante que o primeiro elemento da nova fila continua a ser o elemento que era o primeiro da fila antes da alteração desta, ou seja, que a inserção não foi feita no "topo" mas na "cauda", dada a intuição de que uma fila é uma estrutura linear com dois extremos para operação.

Esta concentração do método no comportamento observável através de um conjunto de propriedades, bem como a inexistência de um estado explícito, tem a vantagem de permitir que se definam TAD em completo afastamento de preo-cupações de implementação e, portanto, de forma bastante abstracta. A *reescri-ta* será o mecanismo fundamental para que se realizem cálculos e provas sobre termos, sempre com base nos axiomas.

Em certos casos, porém, a não existência de um modelo explícito torna as especificações complexas. Por exemplo (cf. [Loeckx 87]), a definição algébrica de um *conjunto matemático*, obrigando à garantia de inexistência de duplicados, torna a sua especificação algébrica difícil usando um conjunto finito de axiomas. Uma solução comum para tal tipo de problemas é a consideração de um conjunto de *operações privadas* sobre o TAD, definidas pelo especificador e posteriormente não acessíveis ao utilizador da implementação do mesmo, que funcionam como predicados que garantem propriedades a serem observadas por qualquer implementação do TAD e que, na abordagem algébrica, são designados por *operadores escondidos*[31], que são o equivalente algébrico aos predicados que na abordagem por modelos, como veremos a seguir, se designam *invariantes*.

A abordagem algébrica favorece, como se pode verificar pelo exemplo dado, níveis de abstracção e de formalização elevados. Do ponto de vista da utilização prática das especificações algébricas, ou até do seu ensino, este elevado grau de formalização tem implicações no, por vezes, elevado conjunto de conhecimentos de matemática e lógica que têm que ser invocados.

Nenhuma linguagem de base algébrica até agora surgida trata questões temporais, dado assentarem numa base funcional. Problemas relacionados com a modularidade, a parametrização e o refinamento de especificações

[31] *hidden operators.*

algébricas têm vindo a ser abordados com sucesso em sistemas tais como ML [Harper 86] ou OBJ [Goguen e Meseguer 82], com implementações disponíveis, ou ainda no sistema Larch [Horning et al. 85] ainda em desenvolvimento. O tratamento de erros e a expressão de falha de comportamento são em geral de tratamento complexo na abordagem algébrica. O problema reside no facto de que quando as funções podem ser parciais (i.é. não definidas em pontos dos seus domínios), uma adicional e considerável complexidade notacional tem que ser introduzida. As actuais abordagens vão da pura exclusão da expressão de situações de erro, à inclusão de um elemento genérico de erro (indefinido = ?) ou mesmo, apenas em trabalhos teóricos, à introdução de valores de erro associados às espécies.

Em [Ehrig e Mahr 85] é apresentada uma completa revisão dos conceitos al-gébricos fundamentais ao método.

3.4.2.- Métodos baseados em Modelos.

Na abordagem algébrica, a cada espécie (ou tipo) é associado um identificador, são dadas as assinaturas das várias operações e a semântica do sistema é espe-cificada usando axiomas que relacionam operações entre si. Porém, quando a complexidade de um sistema é grande, apesar da modularidade permitida na maior parte dos sistemas de especificação algébrica, o número de axiomas pode tornar-se bastante grande.

Os métodos *baseados em modelos matemáticos* surgem como alternativa aos métodos algébricos. Nestes métodos, que trazem até ao "software" uma larga tra-dição em ciências exactas e de engenharia, uma especificação é um *modelo ma-temático* explícito do sistema. Este modelo é constituído pelos seguintes compo-nentes:

- uma associação de um *tipo matemático*, dentre os definidos pelo méto-do, a cada espécie de objectos do sistema, passando a ser o seu mo-delo abstracto, matemático, na especificação;
- a identificação de um *estado interno* do sistema, também modelado por um tipo matemático, sendo sobre o conjunto dos seus valores válidos que se especificam as operações definidas;
- cada operação de interrogação ou observação do sistema é especifi-cada como uma função matemática, sem "efeitos-laterais"; contudo permitem-se também operações que, para além de poderem alterar o estado podem também produzir resultados, sendo os argumentos destas operações os tipos matemáticos definidos como modelos dos tipos do sistema; tais operações são especificadas como relações ma-temáticas envolvendos os estados antes e após a sua execução.

Analisemos agora a especificação por modelos do TAD *Queue,* anteriormente especificado algebricamente, para melhor se verem as diferenças entre os dois métodos.

Comecemos por tentar encontrar um modelo matemático que possa ser apropriado à representação abstracta de uma "fila de espera", dentre os que usualmente fazem parte do leque de modelos matemáticos oferecidos por estes métodos, designadamente *conjuntos*, *sequências*, *funções finitas*[32], *tuplos* e *rela-ções binárias*.

O tipo *Queue* representará o estado interno da "fila de espera", e sendo uma fila de espera uma estrutura linear, uma *sequência* será utilizada. Utilizou-se no exemplo um estilo funcional na especificação das operações sobre uma "fila de espera". É ainda de notar que a linguagem matemática usada na especificação das funções tem que ser adequada aos modelos matemáticos que são argumen-tos destas. Escolhida uma *sequência* de objectos para modelar a "fila de espera", não será de estranhar que qualquer manipulação da "fila de espera" deva ser especificada usando as funções sobre sequências definidas na linguagem.

QUEUE
Estado: *Queue* ? Elem*
Elem ? Nat

inv-Queue(q) ? len(q) < 200

opers

initq: ? queue /* *inicialização* */
initq() ? < >

enqueue : queue x elem ? queue /* *inserção* */
enqueue(q, e) ? q ^ <e>

dequeue : queue ? queue /* *retira primeiro* */
dequeue(q) ?
pre-cond: ? empty?(q)
tail(q)

first : queue ? elem /* *determina primeiro* */
pre-cond: ? empty?(q)
head(q)

---

[32] correspondências.

```
        len : queue ? nat                          /* comprimento da
fila */
        len(q) ? length(q)

        empty? : queue ? bool                      /* está vazia ? */
        empty?(q) ? ( q = <> )
```

Foi propositadamente introduzido um predicado sobre os valores do tipo do estado, *Queue*, indicando que nenhuma *queue* válida poderá conter mais de 199 elementos. Este predicado, que determina a gama de valores válidos e inválidos do estado, designa-se *invariante de tipo de dados*, e deve ser satisfeito por todas as operações que alteram o estado, sendo óbvio que a operação crítica é a ope-ração de modificação do estado *enqueue*.

Algumas das funções possuem uma cláusula designada *pré-condição*, que indica que a função não é uma função total, ou seja, definida para todos os valores do seu domínio (argumentos), indicando-se explicitamente em que casos o resultado da função não se encontra definido. No exemplo, torna-se claro que as funções *dequeue* e *first* são indefinidas caso a *queue* parâmetro se encontre vazia (note-se a relação com os axiomas de erro da especificação algébrica).

São estes os principais componentes de uma especificação baseada em modelos. Dependendo dos formalismos empregues, após a construção deste modelo alguns métodos permitem a imediata, ou quase imediata, construção de um protótipo executável do sistema em desenvolvimento, permitindo deste modo que numa fase muito prévia de concepção, esta possa ser iterada perante um modelo ou simulador do sistema, sem que seja perdida a possibilidade de, tal como noutros, se possam realizar provas estáticas de correcção.

A linguagem de especificação CAMILA [Almeida e Barbosa 91], formalismo de especificação por modelos usado nesta tese, é de base funcional e possibilita o desenvolvimento rápido de protótipos dos sistemas especificados. Dada a sua importância pela extensão da sua utilização na apresentação da tese, dedica-se o APÊNDICE A a uma mais completa apresentação das suas características e da sua sintaxe.

Terminada a fase de construção (e possivelmente iteração) de uma especifi-cação formal, segue-se a fase em que, a partir desta, uma implementação deve ser construída. Os passos necessários à passagem deste nível abstracto até ao nível concreto de uma implementação, por *refinamento* ou *transformação*, não são em geral contemplados de forma sistemática.

Excepção deve ser feita ao Vienna Development Method (VDM) [Jones 80] [Jones 90] que, pela sua maturidade e pelo pragmático compromisso entre

rigor e total formalidade, sugere um método eficaz para se atingirem implementações correctas a partir de especificações por modelos.

O método de refinamento é designado por *refinamento progressivo*[33], pois propõe que, em sucessivos passos, os tipos matemáticos usados como modelos na especificação sejam *reificados*, i.é., materializados em representações mais concretas no sentido da implementação, sendo as operações de seguida refina-das de acordo com a reificação de dados, tal como se ilustra na figura 3.6.

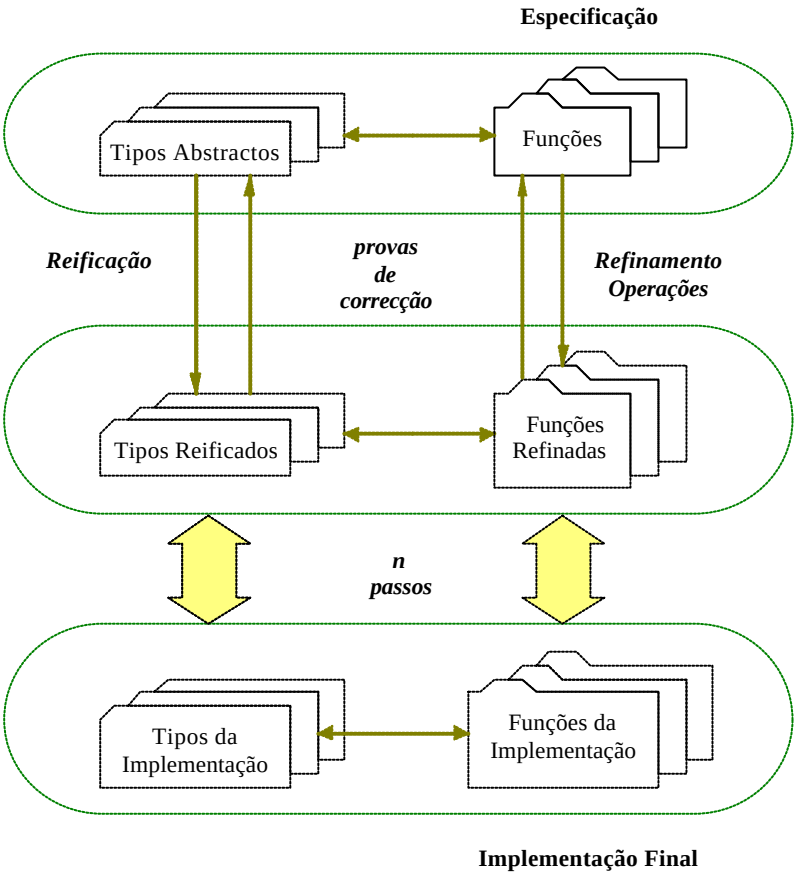

Fig. 3.6. - O Método VDM.

Em cada passo de reificação de dados, escrevem-se as designadas funções de “retrieve” ou de "recuperação de abstracção", que tornam possível provar que todos os valores abstractos são representáveis pelo tipo reificado, designando-se esta prova por "prova de adequação". Outras provas sobre as operações refina-das são realizadas, tais como a garantia da preservação dos invariantes defini-dos a este nível.

[33] *stepwise refinement*.

As provas de correcção em VDM realizam-se usando cálculo de predicados, indução estrutural sobre os tipos matemáticos e um esquema de prova baseado no estilo lógico da "dedução natural". Cada prova a realizar obedece a uma regra de prova bem definida, restando o ónus do tratamento matemático. Um exemplo bastante completo quanto à estrita aplicação do método pode ser encontrado em [Fielding 80].

A linguagem de especificação usada em VDM é a linguagem Meta-IV [Bjørner e Jones 82], linguagem que possui uma semântica denotacional que lhe confere o valor de um formalismo de descrição de significado preciso. No entan-to, a linguagem não é executável, o que impede a construção de protótipos. Po-rém, inúmeras ferramentas foram desenvolvidas quer de apoio às especificações quer de apoio às provas.

Procurando juntar ao rigor do método a possibilidade da criação de protóti-pos, algumas linguagens de especificação por modelos executáveis foram criadas, em particular, *metoo* [Henderson e Minkowitz 86] e, posteriormente, CAMILA [Almeida e Barbosa 91], esta desenvolvida na Universidade do Minho.

Na sua versão actual, VDM é um método bem reconhecido, em particular na área da especificação de sistemas de informação sequenciais, depois de ter sido utilizado no contexto da IBM para a especificação de linguagens como ADA e CHILL, e em muitos outros projectos industriais, encontrando-se em fase de standardização.

Ainda que seja um *método construtivo*, VDM não tem a si associados de base mecanismos particulares para a composição modular de especificações. Alguns esforços nesse sentido têm vindo a ser realizados, sendo de salientar os que têm sido apresentados no âmbito do trabalho para a standardização do método, de-signadamente [Fitzgerald e Jones 90], [Middelburgh 90] e [Fitzgerald 91].

VDM é um método bem fundamentado de especificação por modelos, tendo originado muitas ideias que podem ser utilizadas por outras abordagens seme-lhantes, designadamente quanto à base matemática das provas.

Destas outras escolas de especificação por modelos, seria uma omissão grosseira a não referência à escola de Oxford e à linguagem de especificação Z [Spivey 89] aí desenvolvida. A linguagem Z, procurando ser mais uma linguagem de descrição do que uma base de desenvolvimento por refinamento, coloca uma ênfase maior na prova do que numa possível construção de protótipos, ou até esquema de refinamento. De base lógica, tal como Meta-IV, a linguagem Z oferece um exagerado grau de liberdade quanto à forma de especificação de dados e de operações. Ainda que os tipos matemáticos sejam os mesmos que se apresentaram atrás em relação a VDM, o mecanismo básico de construção de especificações em Z designa-se

esquema[34], sendo usado para representar dados e suas propriedades, ou apenas operações, ou a composição destas seguindo uma *linguagem de esquemas* igualmente disponível [Hayes 87], não existindo claramente a noção de *estado*.

Por exemplo, a declaração do tipo Queue seria neste caso feita usando o es-quema,

```
___ Queue _____________________________
    queue : seq N          /* variáveis */
_______________________________________
    # queue < 200          /* predicados */
_______________________________________
```

no qual se declara a variável *queue* como sendo uma sequência de naturais e no qual, na metade inferior, se descreve num predicado as propriedades a que os valores deste tipo devem obedecer. Este esquema é um definidor do tipo Queue.

O esquema seguinte mostra como se especifica uma operação que, receben-do um argumento de entrada e?, vai alterar o estado da *queue*, cf. ? Queue, e como se especifica na *pós-condição* o novo estado da *queue*, representado pelo identificador *queue´*.

```
___ enqueue ___________________________
    ? Queue
    e? : N
_______________________________________
    queue´ = queue ^ [e?]
_______________________________________
```

Para além deste tipo de especificações baseadas em *pré* e *pós condições*, tal como em VDM, Z permite também especificações puramente funcionais, e até especificações única e simplesmente baseadas em predicados e, portanto, mais orientadas às propriedades.

A linguagem de esquemas permite a comunicação entre esquemas, sob a forma de passagem de determinados dados, quer de modo sequencial quer sob um mecanismo de "piping", sendo no entanto esta comunicação de um para um.

---

[34] *scheme*.

A possibilidade de especificação modular existente em Z é, no entanto, desacompanhada da existência de um método ou, pelo menos, de alguma nor-malização na sua aplicação, pelo que em geral as apresentações acabam por ser realizadas com estilos muito diferentes e nem sempre legíveis, necessitando por vezes da introdução de algumas extensões visando a sua utilização em grandes especificações [Martins 87a].

Os maiores projectos envolvendo especificações realizadas na linguagem Z foram, aparentemente, a especificação do sistema transaccional CICS da IBM, enquanto que VDM foi aplicado, para além da especificação das linguagens re-feridas, na especificação da base de dados relacional System R, igualmente para a IBM.

Em [Monahan e Shaw 91] é apresentada uma interessante perspectiva histó-rica destes métodos (VDM e Z), bem como uma comparação das suas géneses e das suas actuais diferenças técnicas.

3.4.3.- Abordagens Orientadas à Concorrência.

Incluir-se-ão nesta secção referências às principais abordagens disponíveis à especificação de sistemas exibindo paralelismo, concorrência e comunicação nos seus componentes, ou seja, sistemas nos quais existem entidades autónomas, interacção entre estas entidades a qualquer momento das suas execuções, tendo o sistema que gerir comunicação sobreposta no tempo (simultânea), entre os seus componentes.

Em geral, estas entidades são formalizadas como *agentes*, ou *processos*, in-ternamente sequenciais, e comunicantes, associados a uma componente de comportamento dinâmico resultante da sua interacção, por comunicação, numa escala temporal que, mais abstracta ou mais concreta, é incompatível com a visão mais funcional, baseada em transições de estados de tipo atemporal e in-terior oferecida pelos TAD.

A modelação completa de um sistema concorrente composto por agentes ou processos sequenciais comunicantes, envolverá a especificação, a determinado nível de abstracção, de quatro componentes principais: o *comportamento interno* de cada agente, o seu *comportamento externo*, a *estruturação* do sistema em agentes e a *comunicação* entre estes.

O especificação do *comportamento interno* de um agente (processo) consiste da especificação da estrutura do seu estado interno e privado, em termos de um conjunto de atributos; do conjunto de operações de acesso a tal estado; do conjunto de estados internos válidos; do possível conjunto de estados iniciais e das possíveis transições de estado associadas a cada operação definida. Dado o conjunto de estados iniciais e o conjunto das operações, a especificação interna tem implícita um conjunto de sequências possíveis de transições de estado. Por exemplo, nas anteriores especificações de uma *Queue*, não existe nenhuma ope-ração que permita transitar de uma *queue*

com dez elementos para uma *queue* vazia. Tal pode ser inferido das operações disponibilizadas. Por outro lado, foi igualmente especificado que, encontrando-se uma *queue* vazia, qualquer tenta-tiva de realizar o *dequeue* de um elemento não será uma transição aceitável.

No entanto, a expressão destas restrições às transições de estados, ou seja, à sequência de operações, não é realizada explicitamente, sendo, em geral, difícil de ser feita em termos de transições de estados. Assim, uma alternativa consiste em considerar-se que estas operações vão ser executadas em resultado da recepção de *eventos* resultantes da interacção do agente com outros agentes, e assim, restrições de comportamento passam a ser descritas como restrições sobre a sequência de *eventos* aceitáveis pelo agente. A especificação deste com-portamento sob a forma de predicados sobre os eventos em que o agente pode participar, designa-se por especificação do *comportamento externo*, ou seja, ex-ternamente observável. A *comunicação* é o mecanismo através do qual *eventos* são transmitidos entre agentes.

A discretização da dimensão contínua temporal na análise e representação do comportamento de sistemas concorrentes[35], como mecanismo de abstracção, faz com que a unidade atómica, em geral indivisível, a considerar como base de evolução passe a ser o *evento*. Assim, é de salientar que o comportamento global observável de um sistema concorrente vai depender dos comportamentos indi-viduais dos seus componentes, por sua vez dependentes das suas transições internas de estados, por seu lado função do modo como o sistema lida com a potencial simultaneidade da ocorrência de *eventos*.

Mesmo que não entrando em detalhes teóricos, apresentam-se, ainda assim, as principais abordagens à especificação de sistemas concorrentes, ou, se se pretender, de sistemas de processos ou agentes. Duas grandes classes de méto-dos são aqui considerados: *Redes de Petri* e *Álgebras de Processos*.

Redes de Petri.

Redes de Petri[36] são modelos abstractos e formais que representam o fluxo e o controlo de informação num sistema. A teoria das Redes de Petri é devida a Carl Petri que na sua tese de doutoramento [Petri 62] desenvolveu um modelo do flu-xo de informação em sistemas exibindo concorrência assíncrona. São referênci-as obrigatórias sobre este tipo de formalismo os trabalhos de Peterson [Peterson 77] e de Reisig [Reisig 85].

As Redes de Petri têm sido usadas com especial incidência na especificação do comportamento de sistemas reactivos, ou seja, sistemas que

[35] veja-se o que se passa igualmente na área da simulação de sistemas.

[36] *Petri Nets.*

são tipicamente "event-driven" por acomodarem, com facilidade e até naturalidade, as noções de *evento* e *estado*. A noção de *transição de estado* associada à ocorrência de um evento é explícita neste modelo, especificando-se, por cada ocorrência de evento, o conjunto de estados por esta influenciados. Redes de Petri assumem assim um carácter operacional, dado que especificam o sistema pretendido fornecendo um modelo que simula o comportamento desejado deste.

A representação pictórica de uma Rede de Petri consiste num *grafo*, onde dois tipos de nodos podem ser reconhecidos (cf. Fig. 3.7): círculos (designados *lugares*) e barras (designadas *transições*). *Arcos orientados* relacionam lugares com transições e transições com lugares. Os lugares são feitos corresponder a *estados locais*, ou *condições*[37], correspondendo as *transições* à ocorrência de eventos. Estas componentes, desta forma relacionadas entre si, representam as propriedades estáticas dos sistemas.

Porém, as Redes de Petri representam também as propriedades dinâmicas destes, através do posicionamento e deslocação, ao longo do grafo, de marcas, designadas *tokens*, apresentadas como pontos negros, que representam as con-dições, ou estado actual do sistema.

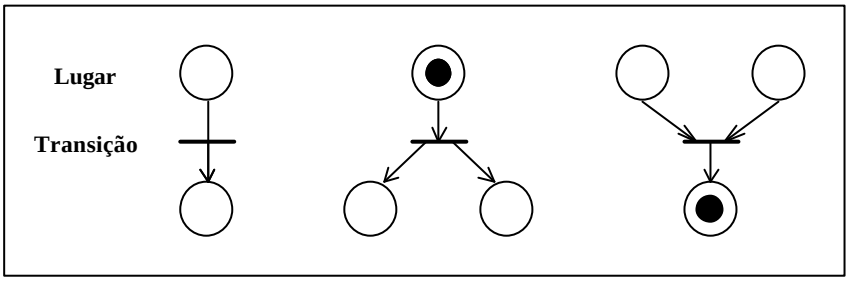

Fig. 3.7 - Esquemas básicos de uma Rede de Petri.

A distribuição dos *tokens* na rede designa-se por *marcação*. A partir de uma *marcação inicial*, o comportamento é representado pela movimentação dos *to-kens* ao longo do grafo. Tal acontece em resultado do *disparo*[38] das transições que se encontrem em condições de o fazer, i.é., habilitadas[39], o que acontece se todos os seus lugares antecessores, ou de entrada, possuirem *tokens*.

O *disparo* de uma transição, ou seja, a sua activação, implica a remoção dos "tokens" dos lugares de entrada da transição e a sua passagem para os lugares sucessores. Em cada momento vários podem ser os lugares marcados, situação que corresponde a múltipla actividade.

---

[37] relacionadas com a possibilidade de ocorrência de determinados eventos.

[38] *firing*.

[39] *enabled*.

Em resultado destas múltiplas marcações, situações de conflito entre transi-ções podem ocorrer (cf. Fig. 3.8), porque várias transições podem estar, num dado momento da execução, simultaneamente habilitadas. A escolha da transi-ção que irá ser disparada é realizada de forma aleatória dentro do modelo, ou por entidades externas não modeladas (por exemplo, um utilizador externo). Assim, uma Rede de Petri modela a actividade do sistema como a que resulta da ocorrência de uma sequência de eventos discretos, cuja ordem é uma das várias ordens possíveis permitidas pela estrutura estática. Em resultado, a execução de uma Rede de Petri assume características de *não-determinismo*.

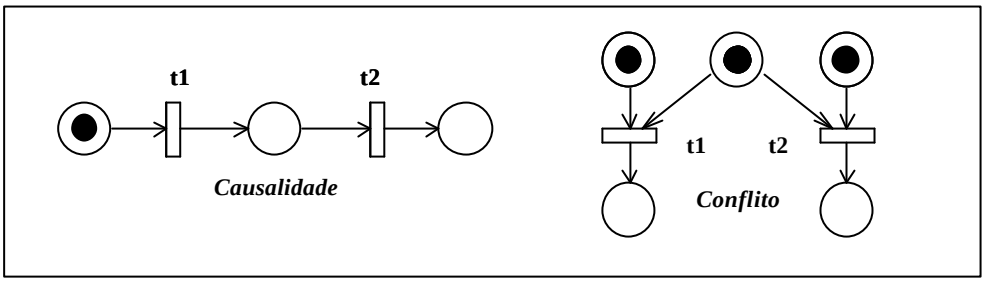

Fig. 3.8 - Causalidade e Conflito em Redes de Petri.

A vantagem de abstracção na modelação que o não-determinismo introduz, tem por desvantagem o aumento da complexidade dos tratamentos formais do modelo. A restrição de que o disparo de uma transição seja *instantânea*, ou seja, em tempo zero, garante a não simultaneidade da ocorrência de eventos, vistos como atómicos ou primitivos e, portanto, não simultaneidade nos disparos das transições.

Assim, o modelo, para além de *não-determinista*, implica *não simultaneidade*. Eventos não atómicos, ou seja, não instantâneos, podem ser estruturados em sequências de eventos atómicos.

Considerando que a rede total é composta pelas diversas partes do sistema a modelar, a natureza *assíncrona* das Redes de Petri torna-se clara quando se verifica que os eventos que dizem respeito a cada componente ocorrem em completa independência uns dos outros. Adicionalmente, a *sincronização* de actividades é também de fácil expressão, sendo criados lugares que estabelecem a ligação síncrona de comportamentos.

Formalmente, uma Rede de Petri R, numa das suas formas mais simples pode ser definida como sendo o tuplo,

$$R = < P, T, F, M >$$

sendo respectivamente, P um conjunto de estados locais, marcados ou não (ou condições), T um conjunto de transições, F um par formado pelas funções de incidência F_i e F_o, representando respectivamente os arcos de entrada e

de saída de cada transição, e M a função de distribuição que associa a cada lugar o número de *tokens* nele contidos.

Por razões que se prendem com necessidades específicas de adequação das Redes de Petri à modelação de problemas com características particulares, são muitas as extensões a este modelo base que têm vindo a ser desenvolvidas. Por exemplo (cf. [Reisig 85]), *Higher Level Petri Nets* procurando resolver problemas de estruturação, *Stochastic* e *Timed Petri Nets* procurando incorporar restrições temporais no modelo e *Predicate/Transition Nets* que incorporam variáveis e pre-dicados que são escritos sobre estas e vão condicionar transições.

As características operacionais das Redes de Petri, bem como a sua sólida e extensa base teórica, aliadas à existência de uma representação diagramática, fazem das Redes de Petri um dos formalismos de modelação de sistemas concorrentes mais empregues em contextos industriais. Inúmeras ferramentas gráficas têm vindo a ser desenvolvidas para que a sua construção seja feita de forma interactiva e validada, bem como outras para a análise e verificação de diversas propriedades dinâmicas dos sistemas modelados.

As Redes de Petri são também bastante adequadas à prototipagem rápida de sistemas, pela operacionalização do modelo que representam, dada a facilidade de interpretação e de execução da representação deste.

As Redes de Petri serão nesta tese utilizadas para a definição da semântica operacional dos *Guiões de Interacção*, o formalismo de especificação de diálogos que irá ser apresentado na secção 6.2. Definida a sua semântica operacional deste modo, as Redes de Petri resultantes da tradução dos *Guiões de Interacção* constituirão a base para a geração de protótipos dos controladores do diálogo assim especificados.

Álgebras de Processos.

Álgebras de Processos são uma outra classe de formalismos para especificação de sistemas concorrentes, cujos exemplares mais referenciados são CSP [Hoare 85] e CCS [Milner 89], de base teórica algébrico-denotacional. A complexidade teórica de uma caracterização formal destes métodos, associada ao facto de não serem estritamente relevantes no contexto desta tese, leva-nos a, não omitindo a sua referência, não realizar a sua apresentação aprofundada.

Álgebras de processos representam os comportamentos de *agentes* (cf. CCS) ou *processos* (cf. CSP) sob a forma de termos de uma álgebra, estendida com duas espécies, representativas de acções ou eventos e de comportamentos, e de um conjunto de combinadores, ou ainda sob a forma de um conjunto de asser-ções sobre os elementos da álgebra.

Diversos modelos semânticos, interpretados em diferentes lógicas, condu-zem a distintas semânticas comportamentais. Estas abordagens possuem uma sólida base formal e tratamento matemático adequado, tendo ainda caracterísi-cas de composicionalidade indispensáveis para a sua utilização em especifica-ções.

A ocorrência de eventos é, em geral, considerada instantânea, não existindo a noção de causalidade entre eventos. A possibilidade de se determinar em que circunstâncias dois processos são equivalentes é, em geral, tratada de forma diferente. Por exemplo, em CSP dois processos são equivalentes se tiverem o mesmo *traço*[40], ou seja, a mesma sequência de eventos até um certo ponto no tempo. Em CCS, no entanto, equivalência comportamental é definida recorrendo à noção de *bissimilaridade*, ou de *equivalência observacional*, que abstrai dos eventos ou acções invisíveis, e considera apenas a igualdade da sequência de eventos que, até um dado instante, foram realizados e dos conjuntos de futuros eventos.

Do ponto de vista pragmático é importante afirmar que o CCS é o modelo de base do formalismo de especificação de protocolos LOTOS [Bolognesi e Briskma 87], enquanto que a linguagem Occam [INMOS 84], linguagem de programação de *transputers*, é baseada no CSP.

De salientar, ainda, que esta perspectiva da estruturação dos sistemas com base em *agentes*, bem como a visão dos seus comportamentos como resultantes da comunicação entre os seus *agentes*, logo com base em controlo distribuído e concorrência, tem sido igualmente utilizada na especificação da estrutura e comportamento de Sistemas Interactivos, em geral recorrendo a CSP para a des-crição da componente comportamental [Alexander 87] [Abowd 91].

Uma abordagem formal aos modelos de agentes muito importante pode ser encontrada em [Henessy 88].

3.5.- O Método de Especificação CAMILA/SETS.

Apresentam-se nesta secção as ideias principais do método de desenvolvimento formal de sistemas "software" designado CAMILA/SETS que, apesar de continuar em desenvolvimento, é actualmente leccionado e empregue no Departamento de Informática da Universidade do Minho.

A figura 3.9 procura ilustrar o Ciclo de Vida do "Software" tal como proposto pelo método. Considerados os *requisitos gerais* do sistema a construir, uma es-pecificação formal em CAMILA/SETS é desenvolvida. Esta especificação pode ser, a um primeiro nível, realizada sob a forma de uma *especificação algébrica equa-cional* ou *teoria algébrica* do problema. Esta especificação, dado o

[40] do inglês *trace*.

seu elevado nível de abstracção, poderá servir para que propriedades dos sistemas possam ser, em abstracto, provadas num contexto algébrico.

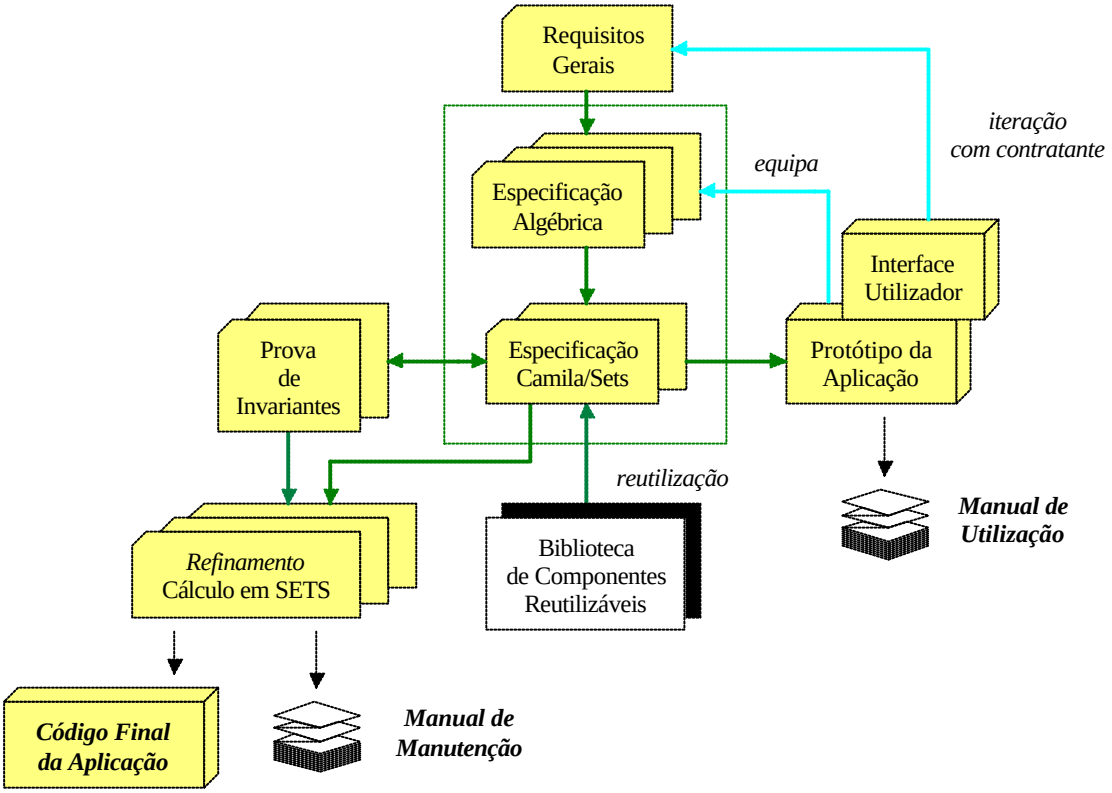

Fig. 3.9 - O Método de Desenvolvimento CAMILA/SETS.

O passo seguinte consiste no desenvolvimento, possivelmente modular, de um modelo matemático dessa teoria algébrica recorrendo à linguagem CAMILA. Tal modelo pode ser total ou parcialmente escrito recorrendo a uma biblioteca de componentes de especificação reutilizáveis disponíveis em catálogo [Oliveira 91].

Esta construção de um modelo abre caminho à geração automática de um protótipo da camada computacional que pode de imediato ser executado e servir para a própria equipa de projecto iterar a sua concepção.

Por outro lado, o método, baseado na tecnologia subjacente, suporta a cons-trução de *simuladores da aplicação*, através da ligação de uma camada interacti-va, entretando desenvolvida, ao código do protótipo da aplicação. De momento, são comuns ligações do código do protótipo a IU escritas quer em VisualBasic sobre Windows, quer em C sobre X-Windows, tendo por base protocolos de co-municação CAMILA/C e C/CAMILA. A linguagem C funciona aqui como linguagem intermediária. A IU é sempre programada independentemente do protótipo, o que se salienta nas figuras 3.9 e 3.10 separando de forma clara os dois módu-los.

Este *simulador da aplicação* poderá agora ser utilizado para se iterar com o cliente quer aspectos de concepção da camada computacional quer aspectos de construção da camada interactiva, o que se torna possível numa fase bastante inicial do projecto, com todas as óbvias vantagens daí resultantes.

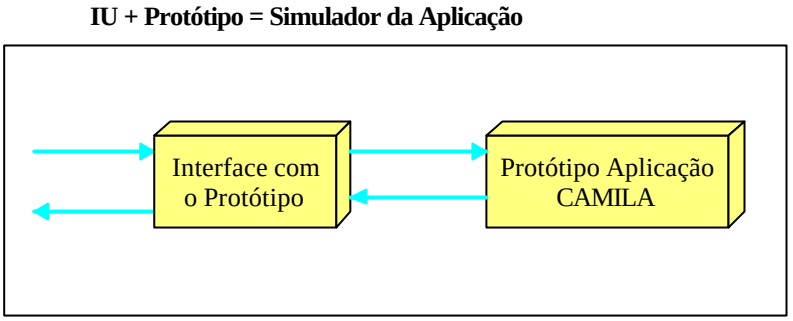

Fig. 3.10 - Ligação Protótipo-Interface.

Após as necessárias iterações sobre este simulador, e fixada com o cliente a funcionalidade final a implementar e até, eventualmente, aspectos de apresen-tação e comportamento da IU, um manual de utilização poderá de imediato pas-sar a ser criado. Numa situação ideal de celebração de contrato, este simulador deveria ser até tomado como referencial para a celebração do mesmo, devendo o sistema final ser com este confrontado em situações de dúvida quanto à satisfa-ção dos requisitos do projecto. Este procedimento poderia trazer à Engenharia da Programação uma credibilidade acrescida, e a ambas as partes, contratantes e contratados, um maior grau de segurança e comprometimento.

Estabelecido deste modo, ideal ou não, o contrato, o desenvolvimento do sis-tema prosseguirá de seguida no sentido da construção de código eficiente para a camada computacional. Partindo da especificação formal construída, a imple-mentação final é *calculada* com base em regras definidas no cálculo de SETS, formalizado e apresentado em [Oliveira 92]. Como este refinamento pode (e deve) ser realizado de forma gradual, poderão existir fases no projecto em que código final coexista com código do protótipo, mantendo-se assim sempre disponível a funcionalidade global inicialmente prometida.

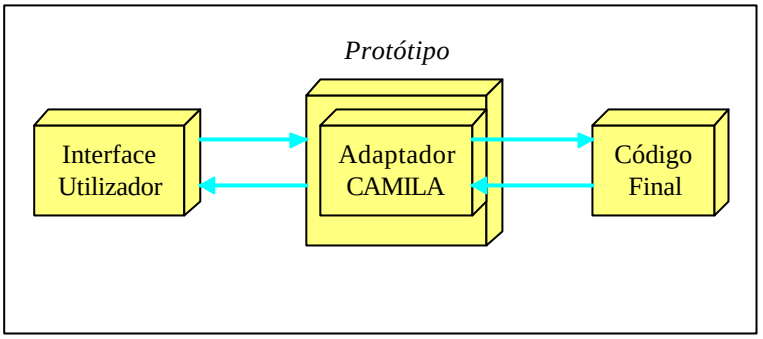

Fig. 3.11 - Implementação Gradual.

Finalmente, o código da IU, anteriormente fixado, é ligado à implementação final da camada computacional, assim se criando o Sistema Interactivo.

Em síntese, o método CAMILA/SETS fornece, para além do rigor que pode ser encontrado em métodos baseados em modelos matemáticos como VDM e Z, an-teriormente apresentados, vantagens adicionais de utilização, designadamente:

- *Possibilidade de criação automática de protótipos, que podem ser mesmo processos UNIX comunicando através de canais bem definidos;*
- *Possibilidade de se criarem, em fases muito iniciais do processo de concepção, simuladores das aplicações, ligando o código do protótipo a uma camada interactiva;*
- *Simuladores que podem, e devem até, ser considerados como referenciais para efeitos contratuais;*
- *Iteração e ciclos de desenvolvimento que podem ser realizados à volta da especificação formal e do simulador;*
- *Cálculo rigoroso e gradual da implementação final, que substitui a tradicional "prova de correcção", suportado pelas ferramentas de construção de protótipos, que possibilitam fases híbridas, ou seja, fases em que o código do protótipo coexiste com porções já construídas da aplicação;*
- *Produção sistemática da documentação de projecto.*

Para terminar, refira-se que ao rigor e sistematização do processo de cons-trução da camada computacional não corresponde porém igual método, siste-matização ou automatização na construção da camada interactiva. Porque esse é o problema para o qual se procuram nesta tese soluções, ainda que com preo-cupações adicionais de generalidade, a abordagem CAMILA/SETS pode ser consi-derada uma potencial "cliente" dessas soluções.

3.6.- Sumário.

Neste capítulo foi feita uma apresentação, ainda que sintética, da evolução das Ciências da Computação, em particular no sentido de prover a Engenharia da Programação com o rigor indispensável à credibilidade do projecto de "software". As principais contribuições para a definição da semântica de linguagens de pro-gramação foram apresentadas. Tendo as linguagens uma semântica precisa, a verificação da correcção de programas passa a ser possível, e as principais abor-dagens com tal objectivo foram referidas, tendo-se salientado a importância das abordagens construtivas. Foi apresentado o conceito de Tipo Abstracto de Dados (TAD), possivelmente uma das noções mais importantes em Ciências da Compu-tação das últimas décadas, com reflexos relevantes quer na prática quer nas "ferramentas" de Engenharia da Programação, designadamente a programação modular e a programação orientada aos objectos, constituindo a base formal do aparecimento de diversos métodos de especificação de sistemas.

A importância da utilização de métodos formais de especificação e desenvol-vimento de aplicações foi salientada, tendo-se ainda apresentado as mais rele-vantes classes destes métodos e seus exemplares. Esta apresentação tornava-se obrigatória dado que alguns destes métodos, ou seus exemplares particulares, são extensivamente aplicados ou explicitamente referidos ao longo da tese.

Pela mesma razão, na secção 3.5 apresentou-se o método CAMILA/SETS, mé-todo que nesta tese se assume como o método utilizado para o desenvolvimento da camada computacional. A linguagem de especificação CAMILA, utilizada para a apresentação de todas as especificações realizadas ao longo da tese, merece, por tal motivo, especial atenção, sendo-lhe dedicado também o APÊNDICE A.