Foundations for a Combination of Heterogeneous Specification Components

Gwen Salaün, Michel Allemand, Christian Attiogbé

IRIN, University of Nantes, France

Abstract
In this paper, we advocate an approach to combine formal specification components. Our work aims at building or reusing specification components, and compose them with a gluing language constituted of a minimal but sufficient set of operators. The glue allows to have at one’s disposal a global formal specification with heterogeneous components as basic entities. The interests are manifold: modelling the different aspects of systems, allowing the use of many existing specification languages, formalizing the links between components in an easy and graphical way, making the reuse of components easier. A case study about a vending machine is specified to illustrate how this approach could be practically used.

Keywords. Multi-formalism Specifications, Heterogeneous Components, Gluing Language, Operational Semantics.

1 Introduction
The use of formal methods is an unavoidable step in the development of critical software systems. Thus, formal specifications follow the analysis of the requirements, and are the basis of the development. An issue is that all the aspects of complex software systems can neither be specified nor verified with only one approach. The joint use of several formal methods, called multi-formalism, integrated or heterogeneous specifications, is necessary for the description of such systems.

On the other hand, the component concept has emerged for some years in several domains. Components take their origin in the software domain in which they represent an independent unit of programming, possibly composed with other ones using interfaces. This idea is recovered here, and is adapted to solve the present issue that is specifying real systems using heterogeneous formal components. We emphasize that in all this paper, components and

1 Email: {salaun,allemand,attiogbe}@irin.univ-nantes.fr

©2002 Published by Elsevier Science B. V.
Salaün

modules are used synonymously (we use modules as a shorthand for specification modules).

Motivations of this work are the following. First of all, the different aspects of software systems have to be specified with appropriate formalisms and have to be formally verified with suitable tools. The use of a single formalism is not sufficient enough to specify the different parts of complex systems. Furthermore, these different formalisms have to be linked in a formal way. Another goal is to simplify the reuse of specification modules written in various languages. This second motivation permits to reuse existing solutions to well-known sub-problems. The work proposed here is based on heterogeneous components, and therefore makes the structuring and the reuse of components easier. This structuring is useful to master the complexity of systems. Finally, we increase the readability of the composition since we choose a linking language with a graphical notation. Indeed, links between components are firstly expressed in a graphical way, before being clarified textually with the complete expression of these compositions.

The basic idea of our proposal is the composition of heterogeneous specification modules. We take one’s inspiration from the approach followed in [2]. This previous study aimed at combining in an heterogeneous manner the process algebra Promela with the language of algebraic specifications Larch. Yet, this work encompasses some limitations, especially due to the lack of formalization of the proposal. In the present approach, we take into account numerous languages compared to the initial work. Our approach is control oriented, and allows at once satisfying motivations of heterogeneous specifications, while constituting an interesting solution for building (and documenting) formal components. The proposal has formal foundations, nevertheless is straightforwardly oriented towards a pragmatic objective.

The remainder of this paper is organized as follows. In Section 2, the formal foundations of our approach is presented into three steps: defining precisely the components as basic entities of the specification, defining the gluing language to connect specification components, formalizing the semantics of this glue. A case study about a vending machine is treated in Section 3. Section 4 introduces related works. We finish with concluding remarks in Section 5.

2 Formal Foundations of our Approach

In the following, languages we wish to compose are split into two main families. Languages are either data oriented or behaviour oriented. We only consider basic languages, i.e. we do not take into account already combined languages such as LOTOS [7]. This is justified because basic languages are more suitable to specify a precise facet of a complex system.

The first family contains formalisms concerning data, i.e. focusing on the static aspects (algebraic specifications, Z [23], B [1]). In the second one are gathered behaviour oriented languages dealing with the dynamic aspects
Concerning data languages, we do not have to face syntactic or semantic inconsistency problems. We assume to have a strong separation between axiomatic oriented (algebraic specifications) and state oriented (Z, B) languages. Indeed, it seems non intuitive, for example, to compose a B module with another one specifying abstract data types. In addition, this restriction is useful in a first step to make the semantics of the gluing language easier. The main problem is at semantic level due to heterogeneous foundations (predicates logic and set theory versus set of algebras). About the dynamic formalisms, while strong syntactic links between modules contents are not allowed, there are no difficulties to connect modules because there is a common semantic model based on LTS\(^2\). A comprehensive study of these heterogeneous aspects can be found in [5].

2.1 Components

We call specification components or modules, a specification written in one of the language previously described. A specification module written in an algebraic specification language can contain the definition of one or several abstract data types. In a similar way, a module written in a process algebra can be composed of several processes. Thus, we consider a general definition of a component, not only containing a single declaration.

A dynamic component contains either process definitions or transition system ones. When we refer to one behaviour of a dynamic module, we consider one of these definitions (e.g. one process or one transition system). This precision is important to simplify the reader understanding of the following semantic definitions in subsection 2.4.

Since our approach is control driven, dynamic components constitute the main behaviour. Therefore, static modules (containing data specifications) are independent, and modules corresponding to dynamic behaviours exploit these data. For the dynamic modules, we have strong links between behaviours and data. These links are mainly expressed thanks to the value passing. More precisely, process algebras handle data at different levels: action parameters, process parameters, and guards or conditional structures. For the basic transition systems, interactions are mainly located in labels of transitions. If more advanced transition systems are used (e.g. Statecharts [12]), links could appear at other levels (e.g. guards). These foundations have been precisely studied in [20], and we consider this formalization in the current approach.

2.2 Operators of the Gluing Language

Now, we deal with the possible links between modules, and we especially detail the glue as a set of operators that we propose to the specifier to connect the

\(^2\) Labelled Transition Systems.
components. These operators are strongly bound to the families of languages to be linked. These gluing constructions are minimal and could be perceived not sophisticated enough. On the contrary, this language is sufficiently expressive for the specifier to connect the specification components as shown in Section 3. For the composition of static components, and the links between both types of components, importation operators are introduced. To compose the dynamic components, we use an operator similar to parallel composition ones met in process algebras.

Our choice about links does not induce a strong separation of concerns, unlike approaches proposed in [4,17] where the gluing principle is based on the synchronized product [3]. However, shortcomings of these works (mainly the difficulty to express the links and a restricted readability) do not appear in our proposal. Now, we start with the different connection possibilities available in our glue: STATIC-STATIC, DYNAMIC-STATIC, DYNAMIC-DYNAMIC. 3

STATIC-STATIC. About algebraic specification modules, we only consider the importation of modules; the operator IMPORT is introduced for this purpose. In a first stage, we avoid giving numerous possibilities of structuring in order to preserve a simple composition language. Concerning Z or B modules, an operator USE is sufficient to describe the use of a module by another one. Data modules (axiomatic or state oriented) are treated separately because their linking operators have different meaning; thus, different names are used to distinguish them.

DYNAMIC-STATIC. For these links, we use an operator NEED to express that a DYNAMIC component needs a STATIC one. Since we work on a control driven specification, a dynamic module needs this construction to interpret data appearing in dynamic behaviours.

DYNAMIC-DYNAMIC. We propose a very general operator of parallel composition named SYN-PC, which induces possibilities of multi-way and synchronous communication. This operator is near from the generalized composition of basic LOTOS or CSP, and has a similar meaning too. It makes it possible the composition of several specification modules, and therefore the composition of behaviours they contain. It has as parameter a set of actions on which modules synchronize themselves. HIDE is proposed as a hiding operator. This construction is useful to hide actions for example within super-components. 4

Renaming. The set of operators contains also the construction RENAME to reinforce the feature of reusing modules (particularly the DYNAMIC ones where data appear). This operator is applied to the module in which the specifier wants to perform renaming of different identifiers.

3 In the following STATIC stands for data oriented modules and DYNAMIC denotes behavioural ones.
4 Component defined by composition of other ones.
Abstract grammar. Now, we precisely formalize the part concerning interconnections of modules, and we summarize in Figure 1 the different linking operators available in our approach.

```
LANGUAGE ::= RE NAMING | CONNECTION
RE NAMING ::= RENAME ID+ ID+ MODULE
CONNECTION ::= STATIC-STATIC-COMPOSITION |
              DYNAMIC-STATIC-COMPOSITION |
              DYNAMIC-DYNAMIC-COMPOSITION
STATIC-STATIC-COMPOSITION ::= IMPORT ASL-STATIC-MODULE ASL-STATIC-MODULE |
                                USE SL-STATIC-MODULE SL-STATIC-MODULE
DYNAMIC-STATIC-COMPOSITION ::= NEED DYNAMIC-MODULE STATIC-MODULE
DYNAMIC-DYNAMIC-COMPOSITION ::= SYN-PC DYNAMIC-MODULE DYNAMIC-MODULE+ ACTION* |
                                HIDE ACTION+ DYNAMIC-DYNAMIC-COMPOSITION
```

Fig. 1. Extended grammar of the kernel

Some nonterminals appearing in the grammar are not detailed at all because they correspond to basic lexical entities. Most of the operators are oriented, particularly RENAME, IMPORT, USE, and NEED. Accordingly, for the importation for instance, there are identifiers which indicate the source module and the target one. Likewise, for the renaming, there are two lists: the first one corresponds to the identifiers to be substituted, and the second one contains new values which replace the previous ones. The renaming applies itself only to one module. Identifiers appearing in lists for the renaming could be terms, operations, actions, events, or others. This vocabulary depends on the language used to write the module where the renaming is done. To remain quite generic, we use the term identifiers. These identifiers are unique inside one module, and made unique (prefixed with the name of their component) at a higher level. At last, we notice that, contrary to the others, the parallel composition operator linking dynamic modules is multidirectional.

2.3 Clarifying some Features

In this section, the goal is to clarify some points and possibilities offered in our approach.

Component characteristics. First of all, in our work components are considered at a specification level whereas, more generally, software components denote a programming level. In software components [24], interfaces are the entrance points to access the component services. Moreover, since the component and its client\(^5\) are developed in mutual ignorance, there exists a

\(^5\) The component which uses the services.
contract to ensure a safe interaction. The environment, in which components evolve, has to provide conditions so that the components can function; there are called context dependencies. Finally, different visibilities of an implementation behind its interface are possible; the main abstractions are whitebox, glassbox, and blackbox.

In this work, interfaces are not necessary since components are either designed by the specifier and consequently are whiteboxes, or reused directly and in this case are considered as glassboxes (this is our choice). Then, the developer can always see the contents of each module (not possible with blackbox abstraction) to get the meaning of the declared behaviours, operations or others, and as a result to connect them. The previous concepts of interfaces, contracts and context dependencies, are not found identically in our proposal, but underlying ideas are captured and managed through the gluing language and its semantics.

Abstract versus concrete components. Following our approach, static and dynamic components are either abstract or concrete. Abstract components contain definitions of data types, operations, or behaviours. While describing a specific system, we model first abstract components with indexed names; then we use concrete components which are obtained by renaming or instantiation from abstract ones. Abstract components without subscript and concrete components are mingled. Furthermore, they could be linked to the abstract data components using the previously defined operators. When concrete data are managed by dynamic behaviours, they implicitly appear in the concrete dynamic components. Figure 2 shows an example of system illustrating this difference of abstraction level. Links involving dynamic components are not expressed using abstract dynamic components, but concrete ones, because we assume that for the same abstract dynamic module (e.g. Dynamici) we could choose different abstract data components to interpret the concrete data terms.

![Figure 2. Example with abstract and concrete components](image)
Super-component. We now discuss the possibility of connecting super-components (or hierarchical components). This possibility is very interesting to structure numerous connected components. It simplifies the representation and makes the hiding of sub-levels details possible. The main idea is that of super-state appearing in Statecharts [12]. Each module can be composed of different sub-components. In case of a connection with a super-component, only its dynamic behaviour could take part into the links (but not the data part). Indeed, the global specification is control driven; consequently behavioural components are visible but not the data ones. Yet, if the specifier needs access to abstract data from a certain module at any hierarchical level, it can model a connection with this module and another one used at a higher level. Care must be taken because this kind of connection between hierarchical levels can harm readability of the whole specification.

Sharing of data. The embedding of concrete data inside dynamic modules is quite natural, and does not preclude, on a wider scale, the sharing of data between dynamic modules. The data to be shared would be managed by a simple process or transition system which provides the minimal behaviour to access and modify the concrete data. Then, the access to the data is done through this dynamic behaviour embedding the data.

Point-to-point versus broadcast communication. Let us consider a case in which two dynamic components are connected using the parallel composition operator of the gluing language. One of the component uses point-to-point communication whereas the other adopts broadcast one. In this situation, the case of one sender and several receivers is taken into account (see the next semantic rules). On the other side, the case of several senders and one receiver is meaningless, and therefore considered neither by our operator nor in the corresponding semantic rules.

Parameterized components. In most languages, especially at programming level, components are parameterized to reinforce their reusability. In our approach, parameterization is taken into account but is not made explicit, i.e. does not appear, for instance, at the definition step (e.g. \( M[P_1, \ldots, P_n] \)). Here, component parameters are particularly used to express genericity possibly appearing in our specification components. As an example, the Figure 3 shows the importation of two data modules (Product and Stock) in a third one in which the stock is explicitly instantiated with the Product data type. Thus, the component defining the stock is parameterized with a generic term, and each component importing Stock is inductively parameterized, unless the parameter is instantiated in the source component.

2.4 Operational Semantics of the Gluing Language

In this part, we formalize the meaning given to each operator of our gluing language. On a wider scale, we detail the semantics of the global behaviour
constituted of linked components. There exist two main kinds of inference rules. The static rules do not induce evolution of the system. This type of rule concerns the renaming operator and the STATIC-STATIC and DYNAMIC-STATIC links. They describe enrichments of context, or constructions of adequate evaluation function. For the DYNAMIC-DYNAMIC compositions, the semantics is seen as a LTS. Inference rules are used to detail the evolution of behaviours for each operator. A LTS is formally defined thanks to a set of states $S$, a set of labels $L$, and a transition relation with type $S \times L \rightarrow S$. The semantics of the dynamic components as well as the semantics of the glue operators linking dynamic components are considered following this abstract definition of LTS.

**Environments.** To define the meaning of grammar operators, we use three environments. The first one is called ASL-E (Algebraic Specification Language Environment), and memorizes informations for modules containing data types expressed with algebraic specifications. This environment is bound to a data module, and is precisely a tuple $<\text{signatures}, \text{axioms}, \text{evaluation function}>$ in which signatures represent operations names and their input/output parameters, and axioms are the well-known algebraic axioms constraining the role of each operation.

The second environment is dedicated to modules written in a state oriented language, and is called SL-E (State Language Environment). It is constituted of a tuple $<\text{signatures}, \text{properties}, \text{evaluation function}>$ in which signatures have the same sense as previously. On the other hand, properties gather different things: variables, invariant, and initialization appearing in B machines or Z schemas. More precisely, these properties correspond to the different pieces essential to describe the state space.

In the large, links between static modules are useful to build evaluation functions which will be used to interpret data appearing in the dynamic behaviours. Evaluation functions are mainly computed from the data definitions (algebraic axioms, or properties for the state oriented languages). This computation is studied thoroughly in the following.

Lastly, the environment $E$ is suitable to dynamic modules to preserve tuples $<\text{signatures}, \text{evaluation function}>$ coming from different importation links between behavioural modules and data modules. Thus, this environment contains operations and their evaluation function. These informations are used to interpret concrete data appearing in the dynamic part. Prefixing with the

---

**Fig. 3. Illustration for parameterized components**
module name could be performed to distinguish two environments in a connection between modules. For example, if $M$ is a module and $DE$ an environment bound to a data module, then $M.DE$ avoids confusion.

**Notations and Variables.** In the following, inference rules are detailed for each abstract grammar rule given in a box. The format used to write the rules is: \( \text{premises} \rightarrow \text{conclusion} \). The \text{label} function returns the identifier of an action. The \text{param} function returns the set of parameters in an action. The \text{Exp}[T/V] logic notation is used to substitute a variable $V$ by a term $T$ in an expression Exp. Here, this notation is extended to perform substitution in a whole module.

Functions \text{in}, \text{out}, and \text{other} verify respectively if an action is an input, an output, or without direction. The \text{asl-ex} function denotes the extraction of signatures and algebraic axioms from a data module written in algebraic specifications. The \text{sl-ex} function denotes the extraction of signatures and properties (variables, initialization, and invariant) from a data module written in a state language. Both previous functions are just parsing functions used to extract desired informations from the different data modules. The \text{∈} function tests if a behaviour is part of a dynamic module. The \text{bound} function denotes that an environment is bound to a module. The \text{eval-extr} function extracts the \text{eval} function from signatures, and axioms or properties. The \text{exist-const} function denotes that there exists a construction (given as parameter) between modules, and this construction is treated to enhance the environment of the source module. Parameters of this function are the name of the construction and the concerned modules.

In the semantic rules, we distinguish some notations in small and capital letters. For signatures, algebraic axioms and properties, symbols in small letters correspond to local definitions whereas symbols in capital letters denote definitions deduced from modules and possible links between them. For example, $\sigma_i$ represents a signature of a single module and $\Sigma_i$ stands for a signature gathering signatures from several modules. The notation for the \text{eval} function is differentiated in the same way. Finally, we gather in Table 1 the variables appearing in the inference rules.

**Inference Rules.** We do not aim at showing the exhaustive list of rules in this paper. We just illustrate the meaning given to some operators of the glue: \text{IMPORT}, \text{NEED}, \text{SYN-PC}. The missing rules can be found in [22].

<table>
<thead>
<tr>
<th>STATIC-STATIC-COMPOSITION ::=</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT ASL-STATIC-MODULE ASL-STATIC-MODULE</td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>$M, M_i$</td>
</tr>
<tr>
<td>$\sigma_i, \Sigma, \Sigma_i$</td>
</tr>
<tr>
<td>$ax_i, AX, AX_i$</td>
</tr>
<tr>
<td>$P$</td>
</tr>
<tr>
<td>$F, F_i, G$</td>
</tr>
<tr>
<td>$\alpha, \beta$</td>
</tr>
<tr>
<td>$A$</td>
</tr>
</tbody>
</table>

The left rule describes the first importation of a module ($M_2$) by $M_1$. The reading here and for the other rules is: if an environment $ASL-E_2$ composed of a set of signatures, a set of axioms, and an evaluation function is bound to an algebraic module $M_2$, and an environment $ASL-E_1$ bound to an algebraic module $M_1$ is empty, and signatures and axioms are extracted from the $M_1$ module, and the evaluation function of the source module is built from signatures and axioms of both modules, and the new $ASL-E_1$ environment composed of the union of signatures, the union of axioms, and the corresponding evaluation function is deduced from the importation, then this environment is bound to the $M_1$ module.

We assume that there are no conflicts in signatures of both modules (empty intersection); care must be taken by the specifier during the modelling. The different names imply different meanings. With this hypothesis, we may have a signature that is the same as another one thanks to a morphism (for instance a renaming); however, in this case, the involved modules are considered different. The evaluation function being one of the result of the importation is computed from signatures and axioms of both modules as explained in the next part.
In the right rule which generalizes the previous one, the \( ASL-E \) environment is not empty because this module has already imported other modules. The new sets of signatures and axioms as well as the new \( eval \) function are computed using suitable sets.

\[
\begin{align*}
\text{DYNAMIC-STATIC-COMPOSITION} &::= \text{NEED DYNAMIC-MODULE STATIC-MODULE} \\
\text{bound}(M_2, ASL-E) &\quad \text{bound}(M_1, E) \\
ASL-E &= <\Sigma, AX, EVAL > \\
E' &= E \cup \{< \Sigma, EVAL >\} \\
\text{exist} - \text{const}(\text{NEED } M_1 M_2) &\quad \text{exist} - \text{const}(\text{NEED } M_1 M_2) \\
\text{bound}(M_1, E') &\quad \text{bound}(M_1, E')
\end{align*}
\]

These rules describe the case in which a data module \( M_1 \) needs another module \( M_2 \) to interpret its local data. The environment \( E \) extracted from \( M_1 \) is enriched with the set of signatures and the evaluation function contained in the \( M_2 \) environment. The \( E \) environment is partially built after the application of the previous rule, and therefore is built inductively. The first step, in which the set \( E \) is empty, is omitted here for the two kinds of data components.

\[
\begin{align*}
\text{DYNAMIC-DYNAMIC-COMPOSITION} &::= \\
\text{SYN-PC DYNAMIC-MODULE DYNAMIC-MODULE+ ACTION*}
\end{align*}
\]

\[
\begin{align*}
F_1 \in_b M_1 \\
F_1 \xrightarrow{\alpha} F_1' \\
\text{label of } \alpha &\notin A \\
M_1' &= M_1[F_1'/F_1] \\
\text{SYN - PC } M_1 M_2 \ldots M_n A &\xrightarrow{\alpha} \text{SYN - PC } M_1' M_2 \ldots M_n A
\end{align*}
\]

The first behaviour of this operator is the possible independent evolution of each involved component. If a behaviour \( F_1 \) appears in a module \( M_1 \), and \( F_1 \) evolves by \( \alpha \) in \( F_1' \), and the label of the action \( \alpha \) is not in the set of synchronization actions, then only the concerned module evolves amongst the set of modules composed in parallel. We note that each module \( M_i \) could evolve in a similar way.
This rule corresponds to the broadcast communication (a sender, and several receivers). The result of the synchronization is the execution of the action $\alpha$, and the evolution of current behaviours in different modules. The test of presence of $\alpha$ in $A$ could be performed with $\beta$ in the same way. A deadlock could be possible if the $n$ processes are not ready for the synchronization. This case is not an error, and is just related to the semantic choice for the parallel composition operator. Another rule, omitted here, corresponds to the synchronization of several processes on a same event (same name) but without direction of communication.

This rule gives the meaning of a parameterized communication between several behaviours composed in parallel. The exchange of data between behaviours is expressed using substitutions. The evaluation function, formalized below, evaluates the parameters of $\beta$. The function selects in the environment.

---

6 We use an intuitive algorithmic notation; parameters $p$ are terms.
ment $E$ (extracted from the module $M_n$ according to the previous rules) the \textit{eval} function to be applied (thanks to the name of the first operation of the term, and after checking its presence in the signatures set appearing in $E$). Then, the current parameter is evaluated using this function. This algorithm occurs for each parameter of $\beta$.

**Algorithm 1** evaluation (param: parameters list; $E$: <signatures, eval> set):

```plaintext
terms list

local result: terms list

init(i)

for each p in param

eval-fct ← find(first-op(p), E)

result(i) ← eval-fct(p)

inc(i)

return result
```

**Computation of the \textit{eval} function.** We have at one’s disposal two types of evaluation function depending on the kind of data specification languages: axiomatic oriented or state oriented ones.

For algebraic specifications, the evaluation function corresponds to a term rewriting function. The rewriting system is obtained from algebraic axioms by applying ordering algorithms, ensuring termination and confluence, as those described in [15]. The rewriting choice is justified since it is suitable to an operational semantics, and accordingly enables us to remain in a pragmatic and executable context. To apply these algorithms, we restrict ourselves to the initial semantics of the data types because the interpretation of a set of axioms by a rewriting system has really sense only in the case of initial model [6]. For algebraic specification languages with \textit{loose} semantics, we restrict them to their initial algebra (e.g. see [14] concerning CASL [9]). Inputs of ordering algorithms are signatures and axioms that are gathered in the $ASL-E$ tuple. Going further, there are more differences in logics than just the initial/loose interpretation problem. For example, an algebraic specification language can use strong equality, another one weak, and a third one existential. If we wish to manage jointly different languages of this kind, we are obliged to prevent the mix of languages based on incompatible logics. Thus, we restrict the composition between these modules to non conflicting logics.

Concerning state oriented languages, the evaluation function is computed from signatures and properties extracted from this kind of module. For each operation name, the function binds the name with the behaviour to apply to the state space which induces modifications on this state space. The state space is characterized by the properties $P_i$ held in the $SL-E$ tuple, that are variables, initializations and invariants. The \textit{eval} function is fully detailed in [22].
Illustration on an Example: the Vending Machine

In this section, we work on the vending machine case study. The system to be built must accept coins inserted by users, and orders of drinks. If the inserted coins are enough, the machine delivers the drink, and gives back coins if necessary. The system is composed of two communicating and concurrent parts: the cash changer and the drink distributer. We assume some restrictions concerning the machine in order to simplify the example: all the drinks have the same price; the cancellation of an order is not managed.

We present in [22] two specifications of the vending machine very similar in their foundations but different enough in the basic languages used to specify components: (1) process algebras and algebraic specifications, (2) labelled transition systems and B. In this paper, we choose process algebras to specify concurrent aspects, and algebraic specifications to model data. In our example, more than two languages are used in the specification. We firstly summarize the data types and the processes modelled for this system, as well as the used formalism: Nat written in CASL, Drink and Stock in Larch, User and CashChanger (CC) processes in CCS, and DrinkDistributer (DD) in CSP. The use of four languages for a such simple specification could be surprising; nevertheless, the single goal here is to illustrate the possibilities of our approach.

The module containing Nat was specified in CASL for a previous work [21], and is directly reused. Modules written in Larch are obtained from a specification treating an invoicing orders system presented in [19]. There are updated by application of the renaming operator RENAME. The next code shows that drinks are deduced from the Product data type, and the stock of drinks is substituted for the stock of products. The first list below contains names to rename, and the second one their substitutes.

\[
\text{RENAME } \langle \text{Product, product, product.ref, eq.product, add.product, remove.product} \rangle < \langle \text{Drink, drink, drink.ref, eq.drink, add.drink, remove.drink} \rangle M_{\text{Product.Stock}}
\]

Now, we show a part of the component obtained after renaming and defining the stock.

```
set name STOCK
declare Sort Stock
declare op
   empty_stock: -> Stock
   add_drink: Drink, Nat, Stock -> Stock
   increase_amount: Drink, Nat, Stock -> Stock
   decrease_amount: Drink, Nat, Stock -> Stock
   in_stock: Drink, Stock -> Bool
...```

127
The CC agent described below in CCS is parameterized with the price of drinks and the number of coins available in the machine. It begins retrieving the sum of money inserted by the user, and the drink to be delivered. Then, CC communicates with the drink distributor to verify if the ordered drink is available, and receives the answer as a boolean. If the drink is available, the sum is sufficient and the cash changer possesses coins enough to give back the money, then it indicates to the distributor that the drink could be delivered, waits for the completion of this action, gives back the money, and increments its number of coins with the drink price.

$$\text{CC} (\text{drinkprice}: \text{Nat}, \text{coinnb}: \text{Nat}) \overset{\text{def}}{=}$$

- getCoin (c: Nat).
- getDrink(d: Drink).
- isAvailable(d).
- availableAnswer(b: Bool).
- (if (b \land \text{drinkprice} \leq c \land (c-\text{drinkprice}) \leq \text{coinnb})
  then deliverDrink(d) . done . giveCoin(c-drinkprice).
  CC(drinkprice, coinnb+drinkprice)
  + if (b \land \text{drinkprice} \leq c \land (c-\text{drinkprice}) \leq \text{coinnb})
  then giveCoin(c) . CC(drinkprice, coinnb) )

The DD process is described in CSP and is parameterized with the stock of drinks that it manages. It has two behaviours. The first one corresponds to the availability of a given drink in the stock. The second behaviour described the delivering of a drink to a user, and the updating of the amount of this drink in the stock.

$$\text{DD} (\text{st}: \text{Stock}) =$$

- isAvailable?d: Drink \rightarrow
  availableAnswer!in_stock(d, st) \rightarrow DD(st)
- □
- deliverDrink?d: Drink \rightarrow giveDrink \rightarrow
done \rightarrow DD(decrease_amount(d, s(0), st))

The User agent is not necessary for the specification of the system. Yet, we insert it so that there are more interacting processes, and not only two agents. Its behaviour consists in inserting coins, ordering a drink, and waiting that the drink is delivered and the coins are given back. In Figure 4, we introduce the specification modules and the links between them.

Concerning these connections, they firstly consist of the importation of the algebraic module defining natural numbers by the module declaring the other data types. After, the dynamic modules need the Drink and Stock sorts, and consequently the module containing them. Finally, modules written in CCS and CSP are composed in parallel with the SYN-PC operator. The link between the Drink and Stock sorts in the Larch module is expressed in the host formalism. It is not written in our gluing language which makes the connection of specification modules possible. Now, we present the textual and comprehensive form of the different links between algebraic modules, and those
Fig. 4. Modules composition for a specification of the Vending Machine

of use between data types modules and the others containing the processes
definitions.

\[
\text{IMPORT } M_{\text{Drink,Stock}}, M_{\text{Nat}} \\
\text{NEED } M_{\text{DD}} M_{\text{Drink,Stock}} \\
\text{NEED } M_{\text{CC}} M_{\text{Drink,Stock}} \\
\text{NEED } M_{\text{User}} M_{\text{Drink,Stock}}
\]

We clarify the SYN-PC composition link between modules written in CCS
and CSP. It is enough to detail modules to be composed and actions on which
processes (defined in the modules) synchronize them.

\[
\text{SYN-PC } M_{\text{DD}} M_{\text{CC}} M_{\text{User}} \{\text{isAvailable}, \text{availableAnswer}, \text{deliverDrink}, \text{done}, \text{getCoin}, \text{getDrink}, \text{giveCoin}, \text{giveDrink}\}
\]

4 Related Works

Our work can be compared with several ones which have the same goal, \textit{i.e.}
combining specification components. However, there are differences in the way
to achieve this goal. Some of these proposals are now introduced. A more
comprehensive comparison with related works is reported in [22]. First of all,
an important reference in software components topics is [24]. A definition
of software component could be the Szymerski’s one: \textit{“A software component is a
unit of composition with contractually specified interfaces and explicit context
dependencies only. A software component can be deployed independently and is
subject to composition by third parties”}. This definition is not kept in our work,
even though general concepts such as reuse, independence, or composition have
been maintained. In our approach, components are the basic entities, but are
specification units and not programming ones.

In the work presented in [18] the authors goal is to specify systems with
state based and event based languages. They especially use Action Systems
Salaün and the CSP process algebra to model the components. Große-Rhode proposes ATS (Algebra Transformation Systems) as formal models of components [11]. ATS correspond to a semantic framework as a common basis in which specifications written in different languages could be interpreted. AltaRica [4] and Korrigan [17] are similar languages, and are especially based on transition systems. Both approaches use the synchronization product of Arnold and Nivat [3] to glue the different specification components.

Numerous researchers work on the development of models, formalisms, and mechanisms to describe integration of heterogeneous components for the modelling of parallel and distributed systems [16]. This approach is based on the concept of coordination. These works are oriented towards programming aspects, whereas we focus on the specification ones. A coordination model is constituted of three parts: entities/components, the media to connect the components, and the semantic framework of the model. This methodological approach has been followed in the presentation of our work.

Finally, over the past decade, software architectures [10] has became an important field of software engineering. Architecture is the organization of systems as a collection of interacting components. ADLs (Architecture Description Languages) are formal modelling notations which focus on the high-level structure of the overall software application. Some ADLs allow the use of formal methods like Wright with CSP, but others posses no formal notations apart from their own-defined one. Finally, formal specifications are able to capture architecture in a similar and more formal way.

5 Concluding Remarks

In this paper, the main purpose is the specification of complex systems with a formal and user-friendly approach. This approach enables specifiers to compose heterogeneous specification components. Components are specified with existing formal languages (algebraic specifications, Z, B, process algebras, transition systems). We propose a simple and minimal language to compose modules written using these formalisms. The operational semantics of connection operators is formalized. A case study is specified as a concrete illustration of this work.

Compared with related works, our approach presents many interests. Firstly, the covering of different aspects (static and dynamic) is achieved thanks to the variety of allowed languages, and their suitability to specify the different involved facets. Besides, this diversity of formalisms provides the specifier with freedom in specification language choices. The reuse of components is simplified too, because links between data and dynamic behaviours are sufficiently generic. The gluing language is based on simple operators with an easily understandable semantics. Finally, readability is improved due to possibilities of graphical representation during the specification steps.

Even though this proposal introduces innovations in the field of hetero-
genuine specifications, this solution is a first attempt in this way. Lots of
difficulties have been overcome, such as semantic rules definition or computa-
tions of $eval$ functions. On the other hand, improvements could be made on
different basic concepts of this work. For instance, the gluing language could
be extended with other operators, such as temporal properties or priorities
between behaviours, to enhance its expressiveness. Another step will be the
strict formalization between behaviours and data inside the components.

The main direction for future works is the further development of the
verification aspects. Thus, our goal is to propose to specifiers a complete
specification environment in which simulation, proofs and testing could be
performed. We do not wish to develop a new toolbox from scratch, but we
prefer to reuse existing tools. Concerning the verification aspects, we espe-
cially aim at proving properties on the global specification using higher-order
tools like PVS [13].

References


Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-
Holland, 1989.

[8] CoFI. The Common Framework Initiative for Algebraic Specification and
Development, electronic archives. Notes and Documents accessible by WWW
and FTP.


---

7 http://www.brics.dk/Projects/CoFI
8 ftp://ftp.brics.dk/Projects/CoFI


