Typing Abstractions and Management in a Component Framework*

Frederic J. Doucet
Center for Embedded Computer Systems
Univ. of California, Irvine
Irvine, CA 92612 USA
e-mail: doucet@ics.uci.edu

Sandeep K. Shukla
Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA 24061 USA
e-mail: shukla@vt.edu

Rajesh K. Gupta
Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA 92093 USA
email: gupta@cs.ucsd.edu

Abstract— We consider the type inference problems in a compositional design environment where the components are automatically instantiated from pre-existing C++-based intellectual property (IP) libraries. We present a component integration language based on scripting for design specification. Our focus is architectural aspects in specification that uses aggregation– as opposed to the more commonly used inheritance– for composition of components. Our approach simplifies architectural specification by employing a type inference and type management environment. We show that the type inference problem is NP-complete. We present a heuristic based on code generation and parameterization to solve the type inference for IP selection in our C++-based composition environment. We have implemented the composition and type management in the BALBOA framework. The results show the utility of our approach.

I. INTRODUCTION

Programming-based high-level modeling frameworks are useful in exploration of complex system architectures. By using programming language to build hardware-software codesign frameworks, the burden on the hardware and embedded system architects shifts from managing design complexity to building architectures, platforms and applications. Both hardware and software can be described in a compositional way. Hardware designs can be conceptualized as a combination of components connected through ports, via signals or channels. Software components can be connected though similar connectors and also through object relations such as aggregation, composition and association [19]. C++ is an increasingly popular choice of language for many such compositional frameworks [3] [2]. C++ is a strongly typed language and therefore components have to be implemented using interfaces that are type compatible. Ensuring such compatibility often results in a significant programming effort that often gets in the way of system conceptualization. Further, conforming to a specific set of interfaces as a requirement for IP manipulation and integration can lower reuse, as the IP is made dependent on its context of usage.

*This work was supported in part by the Semiconductor Research Corporation under a Graduate Fellowship, the Fonds Quebecois de la Recherche sur la Nature et les Technologies under a Graduate Fellowship, the National Science Foundation, the UC Micro Program and Conexant.

The motivation for the BALBOA framework is to provide an efficient and convenient component composition environment that requires minimal amounts of programming effort to build architectural design models [7]. One of the ways BALBOA achieves this is by simplifying the typing requirements imposed on the system architect for the component models in an architecture. Often the designer would leave typing information unspecified or partially specified. In this paper we show that the program of type inferencing is intractable. We then describe a heuristic to solve this problem by applying two type inference rules.

This paper is organized as follows. Section 2 reviews the related work. Section 3 gives an overview of the BALBOA component environment, described the languages, the architecture and the user model. Section 4 describes the type system and the type inferencing in BALBOA, and section 5 describes the implementation of the algorithm.

II. RELATED WORK

System level and hardware specification languages are active areas of research. Most approaches are looking to raise the level of abstraction above the RTL level into either the architectural or the behavioral design space. SYSTEMC [21] and SPECC [9] are examples of approaches that are based on programming languages. There are also a number of component-based approaches, notably COLIF [3] which has shown productivity improvements over RTL approaches, and PTOLEMY which is centered around interaction of models of computation.

The PTOLEMY framework [8] provides a mathematically sound framework for component-based design using an actor paradigm. During the execution of a model, the actors interact according to different models of computation (called domains). Domain-specific directors resolve the domain-dependent interactions, and coordinate the communication between actors from multiple domains. PTOLEMY uses an elaborate type system to statically resolve data types under specified constraints [12] [23]. The static type checking in PTOLEMY can determine the compatibility of a set of component interconnections. PTOLEMY provides polymorphic actors, whose ports can have polymorphic types. The polymorphism of actors is based on a lossless type hierarchy that forms a lattice. Therefore, the static type resolution can be reduced to solution of horn-clauses [18], and thus solved in linear time [23]. However, this requires that actors in the library to conform to the polymorphic actor design principles. This means that types of actors and their ports need to be ordered in the type lattice structure in [23]. This is enforced by requiring interface conformance from new actors, meaning that they need to use the PTOLEMY base classes. For building system models, this requirement is a limitation when importing the legacy C++ components into the IP library.

By contrast, in BALBOA there are a few important differences. BALBOA design components are similar to PTOLEMY actors since the type system has a similar static inference on data types of ports. However, the BALBOA does not explicitly support multiple models of computation, as they are parameterized from the underlying libraries. Second, we are constrained by third party interfaces which may not map to any systematic type lattice. This is because BALBOA does not require components to inherit from a set of base interfaces, but rather use an interface description language to adapt the environment to new components to be placed in the library. Also, the type resolution problem being addressed in BALBOA is a more general problem to which fix-point computation over a semi-lattice can not be applied. In fact, as we show later, our type determination turns out to be an NP-complete problem, and lacks the property of having a "unique most specific type" solution.

As said before, we can introduce new interfaces in BALBOA by using an interface description language. What enables the utilization of components with non-conforming interface in BALBOA is a technique called split-programming. This technique refers to system construction using a combination of a scripting and a compiled language with a common class hierarchy. In split-programming, class hierarchies are used in two programming environments with hooks that enable their combined manipulation [1]. The NS (network simulator) [15] uses a split-programming model built on scripting to create a network simulation environment, with two layers of programming facilities: one for building network objects/components with OBJECT TCL (OTCL), and the other for composing them with C++. Scripting has been used for many years for software component integration in CAD frameworks. Ousterhout argues that a script language interpreter for component integration is essential for API abstraction and reuse because it reduces type dependencies [16].

Script interfaces for C++ components can be generated using a wrapper generator such as SWIG [4, 20]. However, there are a number of limitations when it comes to typing. Type parameterizations and subtyping strategies are not well considered, there are no interpreted type construction facilities. Complex hierarchies are difficult to navigate and manipulate. And also, there is no clear separation between a component and its environment. Sometimes components are not only simple interfaces to classes, but they also have complex internal architectures that separate the environment interactions from the behavior or meaning of the component [13]. In those cases, it is difficult to work with SWIG and other wrapper generators. This is why we decided to define and use the BALBOA interface definition language.

Sometimes, script language can be categorized as architecture description languages, or maybe more appropriately module interconnection languages. Like a script language, an architecture description language (ADL) provides an abstraction of many tedious syntactical details of programming language that are independent of the nature of a system model and not needed for composing a structure. In the software engineering research, there are many ADLs [14] to solve typing mismatch in architectures [11] and to perform model analysis. In the system-level design space, they are often focused on specialized tasks [22], and interoperability can be difficult to achieve [6] when the ADL abstracts only part of the underlying modeling infrastructure. In BALBOA we use a component integration language (CIL) which is similar in spirit to an ADL, but it is not interpreted and not declarative.

III. THE BALBOA COMPONENT ENVIRONMENT

The BALBOA component environment is used to build system models with an architectural perspective. The particularity of the environment is the split-level model with type abstraction and inference. The environment implements a split-programming model like NS, but has more layers and an intermediate language between the script language and the C++. Figure 1 shows the layers in the environment. Languages are on the left side and the run-time structure on the right side.

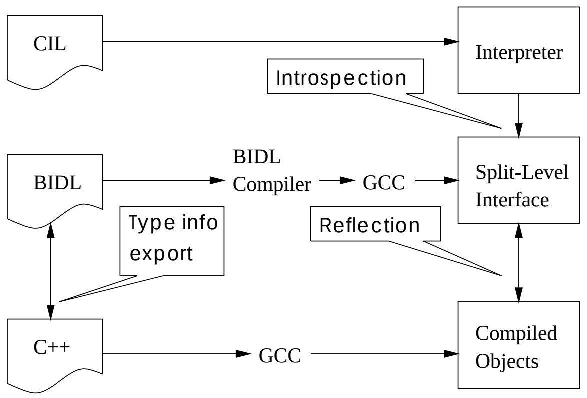

Fig. 1. Layering in the BALBOA environment: languages on the left side, run-time structure on the right side

The highest layer is the scripting layer. An architectural structure is assembled from components by using the component integration language (CIL). At this level, the architect instantiates components, connects them, and inserts test benches, monitors etc. We also call this layer the interpreted layer.

The lowest layer is the component library layer. This layer consists of IP libraries of C++ classes/objects. Theoretically, this layer can accommodate C++ IP models in a wide range of libraries without affecting the implementation of the other two layers. We also call this layer the compiled layer.

The intermediate layer is the wrapper layer. The C++ objects from the compiled layer are contained and manipulated by split-level interfaces (SLI). This layer connects and provides the mechanism for manipulation of compiled objects from the scripting layer. The split-level interface dissociates the modeling interface from the implementation interface.

The split-level interface implements the reflection and the introspection capabilities of the environment. The reflection is the capability of the split-level interface of a compiled object to explicitly captures the attributes, methods, and properties of the object. Introspection is the capability of the CIL language to query the reflected information of a component. This information is described using the BALBOA interface description language (BIDL). This information describes the modeling semantics of the component, and we use this as the "type" of the component. The BIDL was first inspired by the CORBA IDL [5], but it has been customized and extended for the requirements of system-level modeling. The BIDL compiler converts the components' interface description (its type information) into a format that the interpreter can understand. The BIDL is a little bit similar to the CPP preprocessor because it can be thought of expanding the type information, but what it actually does is to customize a split-level interface for the new component.

In BALBOA, when we refer to a component, it means the whole right side of Figure 1. In the CIL a component is a OTCL object with methods and attributes, and this object is shadowed by a number of compiled C++ objects. Figure 2 shows the internal architecture of a BALBOA component.

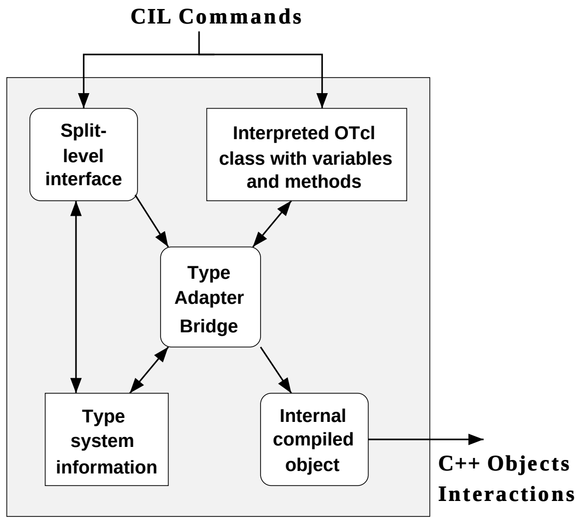

Fig. 2. Internal architecture for a BALBOA component

The internal compiled object is a C++ object that can be a SYSTEMC object, or an object using any other library. We sometimes refer to the internal compiled object as the compiled component. The split-level interface executes the commands in the scope of the components and maintains the type information, which is the meta information about the type of the internal object and its modeling semantics. This implements the introspection and the reflection, where structural information (which attributes and methods a component has) can be queried, like in the Java reflection packages. However, the type information also contains non-functional properties, as for example, system level characterizations such as "functional", "behavior", "structural", "port", "signal", "module" and "process" that are independent of the C++ interfaces. These properties are specified (declared and mapped to C++ classes) through the BIDL, to be instrospected through the CIL.

IV. TYPE RESOLUTION IN BALBOA

The type abstraction means that an interpreted component can be shadowed by a number of compiled components. The C++ library components are often implemented using templates, and the ports of the components may carry different concrete data types. For example we may have an adder which may have various implementations in the library. It may have two input ports which may take floating point numbers as input, and an output port for a floating point output, and another port of Boolean type to indicate overflow/underflow. Another implementation of the adder may take 16 bit integers as input and produce 16 bit output and one Boolean output for the carry.

When building a system, often a designer may not want to make specific type choices for all its ports since that very much depends upon the kind of component to which an instance may be connected to. Using the CIL, the designer can declare a components in an architecture without specifying the input types, or output types; or a designer might choose to explicitly specify the types. If the types are unspecified, but this adder is then connected to a multiplier, whose input types are specified, the environment can infer the types of the adder ports, and the system will then instantiate the appropriate component from the library. This flexibility for designers to be oblivious to C++ typing at the hardware design level is necessary for providing the system designer with a degree of flexibility, and for automatic exploration of alternatives for best design with the available components in the library.

In BALBOA, the type information for each class of components is abstracted in the BIDL. The component library designer, while exporting the component interface into the BIDL, must also specify what different types are available for the specific components. For example, in the case of an adder, the library designer must specify that the adder has been implemented for float, 8, 16, 32 bit integers, single bits and so on. Now, the introspection facility in BALBOA can let the environment and the architect know about such availability of different types. But the designer may choose to let the system infer the appropriate type as well. When the designer chooses to instantiate components without specifying the types for each port, and/or signal, we call it a component with a "partial type" because it may still make sense at the CIL layer.

A. Split typing

In the BALBOA environment, this partial type can also be viewed as "split type" because it is managed through the split-level interface. A component can have a type at the interpretive layer, and multiple possible implementation types at the compiled level. In other words, for a component, there is an interpreted type, and a compiled type. In the adder example, at the interpreter level, all the ports of the adder are just "Port" type. Based on the containment hierarchy and on connections to neighboring components, type inference will associate specific C++ data types to the ports.

In order to illustrate the algorithms for incremental type inference, we use the following notation. Let T be a set of all concrete C++ data types, let P be a set of ports and S be a set of signals in the design created in the CIL. Ports and Signals are abstract types, and at the compiled component level each port type and signal type must be mapped to a concrete type in T. Each port or signal is associated with a data type via mappings, dt_p, and dt_s, such that $dt_p : P \to T \bigcup \{\perp\}$, and $dt_s : S \to T \cup \{\perp\}$. A port is untyped if $p \in P : dt_p(p) = \perp$, and a signal is untyped if $s \in S : dt_s(S) = \perp$, where $\perp$ denotes the fact that no concrete type has been specified or inferred yet. When the type inference is done, dt_p and dt_s must not map any element to $\perp$, and if that cannot be achieved the type-inference algorithm must detect and report that.

B. Components

In the interpreted layer, (i.e. in the CIL) a component c has a set of ports denoted by $ports(c)$. If n is the number of ports for a component, they are denoted by $p_{c_1}, p_{c_2}, \ldots, p_{c_n}$. In the compiled domain, the component c, might be "polymorphic" in its port types, and hence there may be various compiled versions of c, such that the ports of c might have different types from T. The function dt_p has a limited choice in assigning $p_{c_1}, p_{c_2}, \ldots, p_{c_n}$, to the different compiled types.

One can view this as a choice of assignment to a vector of ports $\triangleleft p_{c_1}, p_{c_2}, \ldots, p_{c_n} \triangleright$, from one of the possible rows from a type availability table of ordered rows $T_c \subset T \times T \times T \times \cdots \times T$, each row corresponding to a compiled version of c. In fact, the author of the component library explicitly puts all the possible type combinations for component c's instantiation hence the table T_c through BIDL descriptions. Since signals are used to link a port of one component to a port in another component, if a signal s does not have a type specified by the designer, it can also be assigned a concrete type from a set of types. However, a signal s must be assigned the same type as the ports it is connecting, hence, signals can be thought of as constraints in the port assignment problem.

Definition *The* BALBOA *type inference problem: given a design with a set P of ports, and S of signals, and given the partition of P, into k disjoint sets (where k is the number of components, and components do not share ports): For each component c, with its port vector $\triangleleft p_{c_1}, p_{c_2}, \ldots, p_{c_n} \triangleright$, assign a row from its type table T_c, such that if there is a signal $s \in S$, which connects a port p_{c_i} in component c to another port p_{d_j}, of a component d, then the type assigned to p_{c_i} and to p_{d_j} must be exactly the same.*

Theorem *The* BALBOA *type inference problem is NP-complete.*

Proof: given a type assignment for all the ports, it is straightforward to verify from the type tables, that the assignments are correct, and the ports connected via signals have the same type. Hence the problem is clearly in NP.

For the NP-completeness proof, we reduce the problem of One-in-three monotone 3SAT [10] to the BALBOA type-inference problem. 3SAT is the following problem: given a set U of Boolean variables, and a collection of disjunctive clauses over U, such that each clause is a disjunction of exactly 3 literals, find if there is a truth assignment of the variables in U, such that all the clauses are satisfied. One-in-three 3SAT is a special case of 3SAT problem, where the truth assignment of U has the restriction that if x_i and x_j are two literals appearing in the same clause, then both cannot be assigned the truth value of 1. Finally, the monotone One-in-three 3SAT has the further restriction that no negated literal appears in any clauses.

Given an instance of monotone One-in-three 3SAT, for each clause c, which has three literals $x_{c_1}, x_{c_2}, x_{c_3}$, one can create a table T_c, with exactly 3 rows $\{< 1, 0, 0 >, < 0, 1, 0 >, < 0, 0, 1 >\}$, and given the set of vector of variables in c, $\triangleleft x_{c_1}, x_{c_2}, x_{c_3} \triangleright$, one has to assign one of the rows in table T_c, with the restriction, that if the variable that appears as x_{c_1} in c, also appears as x_{d_i} in another clause d, then the choice of the rows from T_c and T_d must be such that the assigned truth values are the same for x_{c_1} and x_{d_i}. This is a version of the BALBOA type inference problem, and hence BALBOA type inference problem is at least as hard as the monotone One-in-three 3SAT, which is known to be NP-complete [10]. ■

It can be shown that the type inference problem can be formulated as a constraint satisfaction problem such as a relational constraint problem. The BALBOA type inference can be cast as a query evaluation in a relational database comprised of all the type availability tables in the type inference.

V. TYPE INFERENCE IMPLEMENTATION

The type inference framework has three parts: the BIDL language and the BIDL compiler, the run-time infrastructure that captures all the type information, and the heuristic that do the type management, propagation and inference for a design architecture. The BIDL compiler parses a component description and searches for type parameterization relations (TPR), where types parameters are shared between two components. There are two cases of TPR: (a) through types of method arguments (b) through types of the attributes of a component.

A. Type Parameterization Relation through Methods

Consider the following BIDL listing where an `Inport` component type with a `bind_to` method is declared:

```
Component Inport {
  PARAMETERS {type DATATYPE};

  void bind_to(const Signal<DATATYPE>& sig);
};
DEFINE_PARAMS {unsigned int} FOR Inport;
DEFINE_PARAMS {bit_vector32} FOR Inport;
```

The partial type of the component is `Inport`, with a `DATATYPE` type parameter. The last two lines of the listing define two implementations of the component, each one specifying a type parameter value, one for an unsigned integer and another one for a 32 bit vector. In this case, the `Inport` component is shadowed by a C++ class also named `Inport`, whose instance is used for the internal object. When an `Inport` component is instantiated, the type parameter `DATATYPE` is undefined, and the internal object not allocated until there is a valid mapping.

In this listing, the BIDL compiler finds one type parameterization relationship between the argument `sig` of method `bind_to` and the `Inport` component, sharing parameter `DATATYPE`. When the `bind_to` method is called on a inport component, the run-time environment can inspect the full type of `sig` parameter for the value of `DATATYPE`. If it is defined, it can be propagated from the `sig` parameter to the inport component, or vice-versa.

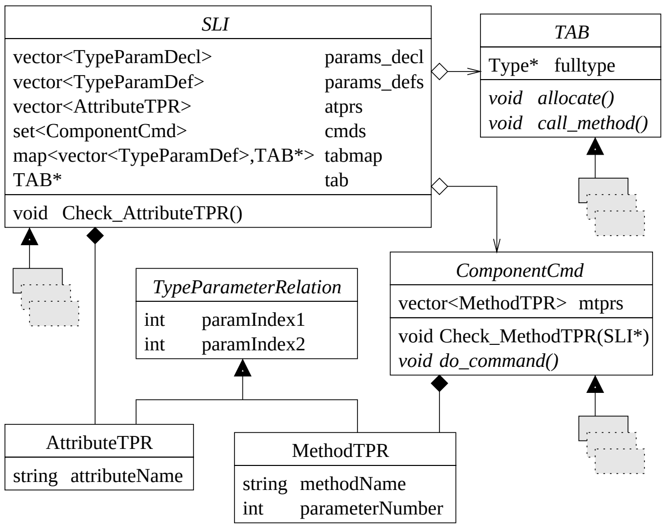

Fig. 3. Type parameterization framework: the split-level interface (SLI) contains the type parameterization relationship by composition (through attributes) and the commands contain the type parameterization relations through methods. Classes in gray are extension generated by the BIDL compiler when adding new component types.

B. Type Parameterization Relation through Composition

Consider the following BIDL listing where a `Buffer` component type is defined, which has two ports `in` and `out`:

```
Component Buffer {
  PARAMETERS {type DATATYPE};

  Inport<DATATYPE>  in;
  Outport<DATATYPE> out;
};
DEFINE_PARAMS {unsigned int} FOR Buffer;
DEFINE_PARAMS {bit_vector32} FOR Buffer;
```

This component is also parameterized, and has two implementations, one for unsigned integer and one for 32 bit vector. The BIDL compiler finds two type parameterization relationships in the listing. The first one is between the attribute (or sub-component) `in` and the `Buffer` component where they share the parameter `DATATYPE`, and the second one is between the attribute `out` and the component also with `DATATYPE`. Type parameter definitions can be propagated to the `Buffer` component along the composition relation when the type parameter of a sub-component `in` or `out` is defined, or vice-versa.

C. Run-Time Structure

Figure 3 shows the simplified class diagram for the run-time environment, in regards to the typing framework. There are 6 classes in the figure and three of them are used as base classes for extensions generated by the BIDL when adding new component to libraries. Those classes are in gray boxes on the figure. The first and most important one is the `SLI` class that implements the split-level interface. The BIDL compiler do specialize this class by inheritance into a class specific for a new component. A specialized class contains the information about type parameters declarations, and at run-time about defined type parameters for an instance of the component. It also contains a vector of attribute type parameter relations `atprs` for parameters shared in composition. There are two methods to check the parameterization relations, we will explain them later in the inference description subsection.

The second class that gets extended by the BIDL compiler is the Type Adapter Bridge (`TAB`) class. This class manipulates the exact type of the C++. Only the TAB knows the specific type and it is the only access to the compiled internal object. The type availability table is implemented through a table of TABs, and it is indexed by the type parameter definition vector `SLI::params_defs`. A specific TABs is derived by the BIDL compiler for every available C++ class usable as the internal object. When all parameters are defined, the `params_defs` vector is set and it can index the TAB map to find the specific TAB for the exact C++ type match.

The third important class is `ComponentCmd` that executes a command on a component. This class is also specialized by the BIDL compiler to implement method calls to the internal object (such as a call to the `bind_to` method of the Inport class). The `ComponentCmd` class contains the type parameterization relations between command parameters and the component.

Finally, note that type parameter relations are tuples $(index1, index2)$ of shared type parameter indexes in the type declaration vectors of two components.

D. Run-Time Type Inference Heuristic

The run-time type inference has two tasks: verify if types are compatible when connections are set and propagate parameters along type parameterization relations. A type parameter can be set by an inference along a relation, but it can also be constrained by the user of the system. The run-time infrastructure also has the task of allocating the internal object when all type parameters are valid and have a single match in the type availability table. If the types are unknown or incompatible, the infrastructure can delay the allocation of the internal object to some other time in the future. In any case, if the type parameters of a component are not all defined, the execution of a command on that component has to be delayed. This is because the full type of the compiled object is unknown; hence we cannot allocate it and invoke its behavior.

There are two ways to do the type inference: do it once when the architectural topology is fully specified (on an elaboration phase), or do it progressively as the topology is specified (like in ML [17] by progressively applying type reduction rules). In this case, parameters can be defined multiple times for a component; therefore commands have to be delayed until the subtypes definitions vector is committed. Note that establishing a connection is a command. There are two policies for the allocation of the internal object of the component. The run-time inference engine can be opportunistic and do the allocation as soon as it is possible– when the type parameter definition vector is defined and there is a type allocation table match. The environment can also be lazy and wait until the user of the system issues a command to allocate the objects. The system will then do the allocation, but it will not run the delayed commands until the user also instructs the system to do so.

The heuristic for type inference is presented in figure 4. When a command is invoked on a component, like connecting a component to a signal, the Run_Cmd procedure will verify the type parameter relation between the component and the command arguments. If all internal objects are allocated, the command can be executed, else it will be delayed. The architect can use the Run_Cmd and Run_Delayed_Cmd_For_Component procedures to propagate type parameters between components. The latter re-runs commands that were delayed because of undefined parameters and delayed allocations. Procedures Check_MethodTPR and Check_AttributeTPR respectively check for type parameterization through methods and compositions. The method propagation procedure will propagate from arguments to component, and then the other way around. The composition propagation procedure will propagate the parameters “downwards” and then “upwards” in the hierarchy.

The Propagate_Type_Param procedure is recursive because it will always check for composition relations which can require more propagation. It also calls the procedure to run the delayed commands for a component to propagate along connections. This procedure is static in the run-time environment. The method propagation procedure is implemented in the `ComponentCmd` class of figure 3, and the composition propagation is implemented in the scope of `SLI`.

```
Run_Cmd ( component, cmd ):
Check_MethodTPR (component, cmd )
if component is allocated and all arguments of cmd are allocated
    execute component :: cmd
else
    delayed_cmds = delayed_cmds ∪ cmd
endif

Check_MethodTPR ( component, cmd ):
for all mtpr ∈ (cmd.mtprs)
    Propagate_Type_Param(component,
                mtpr.paramIndex1,
                cmd.params[mtpr.parameterNumber],
                mtpr.paramIndex2)
    Propagate_Type_Param(cmd.params[mtpr.parameterNumber],
                mtsr.paramIndex2,
                component,
                mtsr.paramIndex1)
end for

Check_AttributeTPR (component):
For all atpr ∈ (component.atprs)
    Propagate_Type_Param(component,
                atpr.paramIndex1,
                atpr.attributeName,
                atpr.paramIndex2)
    Propagate_Type_Param(atpr.attributeName,
                atpr.paramIndex2,
                component,
                atpr.paramIndex1)
end for

Propagate_Type_Param (component1, index1, component2, index2)
if (component2.param_defs[index2] ≠ ⊥ AND
   (component1.param_defs[index1] == ⊥ OR
    CONTINUOUS_PROPAGATION_POLICY))
    component1.param_defs[index1] ←
        component2.param_defs[index2]
    Check_AttributeTPR(component1)
end if
if (OPPORTUNISTIC_INTERNAL_OBJ_ALLOCATION AND
   ∀i: component1.param_defs[i] ≠ ⊥ AND
   tabmap[component1.param_defs] ≠ ⊥)
    allocate internal object of component1
end if
if CONTINUOUS_PROPAGATION_POLICY
    Run_Delayed_Cmds_For_Obj (component1)
end if

Run_Delayed_Cmd_For_Component (component)
for all delayed_cmd ∈ delayed_cmds
    Check_MethodTSR(component, delayed_cmd)
    if component is allocated and all arguments of cmd are allocated
        execute component :: cmd and remove from set delayed_cmds
    endif
end for
```

Fig. 4. Type inference and management heuristics for the BALBOA run-time environment: type parametrization relationships are checked and propagated through method parameters or through attribute composition.

E. Results

The heuristic is implemented in the BALBOA run-time environment, and used in four designs examples. The first one is a behavioral state machine that is refined to RTL FSMDs, the second one is a simple packet switch that is connected to multiple receivers, the third one is a FIR filter connected to a source and a display, and the last one for a memory hierarchy research platform called AMRM.

In the state machine and FIR example, the CIL was used just to select the right data type of the IP from the library. The environment does the type inference, select a component from the IP library that matches the type constraints, and does the allocation and binds the connections. In the packet switch example, the CIL can be used to build an architecture because the design has a regular structure as shown on figure 5.

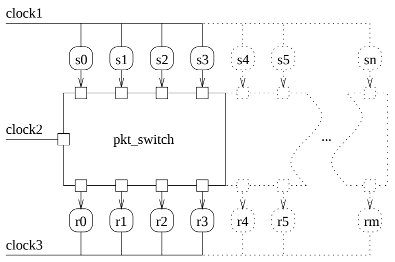

Fig. 5. Packet switch example with a regular structure.

The following CIL segment shows how the topology on the figure can be built:

```
Pkt_Switch pkt_switch
connect pkt_switch.CLK to clock2
for {set i 0} {$i<$NUMBER_OF_PORTS} {incr i} {
  Receiver r$i -id $i
  Signal   pkt_in$i
  Signal   pkt_out$i
  connect  s$i.pkt_out       to pkt_in$i
  connect  r$i.pkt_in        to pkt_out$i
  connect  r$i.CLK           to clock3
  connect  pkt_switch.in$i   to pkt_in$i
  connect  pkt_switch.out$i  to pkt_out$i
}
```

We can also use the CIL to generate and process lists of type-less signals such as:
`set signal_list {in1, in2, in3, out1, out2}`
and connected them through lists of type-less connections such as:
`set conx_list {{a.in1 in1} {a.clk clk} ...}`
In other words, we wrote procedures in the CIL so that components can “understand” how to connect to each other, based on compatible names or on connection templates, and generate the connection code. This shifted the usage of the CIL to an abstraction where the focus is on the interconnections and the structure instead of typing, similar to an ADL. By using the typing abstraction, the complexity of the specifications was simplified for several versions of each design, but it remains to be investigated how this can be used on large designs.

VI. CONCLUSION

This paper addresses an important issue in component-based microelectronic design environments, that is, how to simplify strong typing requirements imposed by the C++-based modeling methodologies and yet maintain the efficiency of compiled simulations. Our approach is to raise the abstraction by split-level programming and automated type inferencing based on the constraints on available specific libraries of components. We have shown that in general this problem is NP-complete. We have presented a heuristic and an implementation that allows for determination of specific port types based on inferences from the overall system architecture, component connectivity or by propagating type information from object to object along sub-typing relationships. While this formulation is useful in component-based designs, it remains to be seen if it can be adopted in practice by system designers who may be oblivious of the details of typing and how it interacts with the overall design methodology. Our future work includes generalization of the type parameter framework to include behavioral typing, to see how compatible two different IPs are, and what is needed to substitute one by the other.

VII. ACKNOWLEDGMENTS

The authors would like to acknowledge Jean-Pierre Talpin and Cristiano Pereira for fruitful discussions about type systems.

REFERENCES

[1] L. Breslau, D. Estrin, K. Fall, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer*, May 2000.
[2] J. Buck and R. Vaidyanathan. Heterogeneous modeling and simulation of embedded systems in el greco. In *Proc. Int. Workshop on Hardware/Software Codesign*, 2000.
[3] W. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A. A. Jerraya, and M. Diaz-Nava. Component-based design approach for multicore socs. In *Proc. IEEE/ACM Design Automation Conf.*, 2002.
[4] P. Chen, D. A. Kirkpatrick, and K. Keutzer. Fast Integration of EDA Tools and Scripting Language. In *IEEE/DATC Electronic Design Processes Workshop*, 2001.
[5] CORBA website http://www.corba.org.
[6] F. Doucet, R. Gupta, M. Otsuka, P. Schaumont, and S. Shukla. Interoperability as a Design Issue in C++ Based Modeling Environments. In *Proc. Int. Symposium on System Synthesis*, 2001.
[7] F. Doucet, S. Shukla, M. Otsuka, and R. Gupta. An Environment for Dynamic Component Composition for Efficient Co-Design. In *Proc. Design Automation and Test in Europe Conf.*, 2002.
[8] Ed Lee et al. The Ptolemy Project. Home Page: http://ptolemy.eecs.berkeley.edu/.

[9] D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, 2000.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, November 1995.

[12] E. A. Lee and Y. Xiong. System-Level Types for Component-Based Design. In *First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*. Springer, October 2001.

[13] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proc. 4th ACM SIGSOFT Symp. on Foundations of Software Engineering*, 1996.

[14] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Trans. on Software Engineering*, January 2000.

[15] NS: The Network Simulator home page: http://www.isi.edu/nsnam/ns.

[16] J. K. Ousterhout. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer*, March 1998.

[17] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[18] J. Rehof and T. Mogensen. Tractable Constraints in Finite Semilattices. In *Third International Static Analysis Symposium*, volume 1145 of *Lecture Notes in Computer Science*. Springer, September 1996.

[19] V. Sinha, F. Doucet, C. Siska, R. Gupta, S. Liao, and A. Ghosh. YAML: A Tool for Hardware Design Visualization and Capture. In *Proc. Int. Symposium on System Synthesis*, 2000.

[20] Simplified wrapper and interface generator (SWIG) home page: http://www.swig.org.

[21] SystemC. OSCI. Home page: http://www.systemc.org.

[22] H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Architecture Description Languages for System-on-Chip Design. In *Asia Pacific Conference on Chip Design Language*, 1999.

[23] Y. Xiong and E. A. Lee. An Extensible Type System for Component-Based Design. In *The 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*. Springer, April 2000.