Applying the Itacio Verification Model to a Component-Based Real-Time Sound Processing System

Agustín Cernuda del Río, Jose Emilio Labra Gayo, Juan Manuel Cueva Lovelle

Departamento de Informática
Universidad de Oviedo
C/Calvo Sotelo S/N
33007 Oviedo, Spain
+34 985 10 {50 94, 33 94, 33 96}
{guti, labra, cueva}@lsi.uniovi.es

Abstract: The goal of the Itacio component model is to statically verify software systems made up of components. It relies on Constraint Logic Programming for stating the requirements and guarantees of each component, and offers a model of verifying that a system built by combining components fulfils the requirements for their proper operation. Itacio is driven by the goals of static, automatic verification and high feasibility, including an easy adoption by developers and the use of well-known technologies.

The notion of component in this model is deliberately open; thus, this method can be (and has been) applied at different abstraction levels in the development process. Among them, one obvious use (and its original motivation) is the description and verification of software components in the most usual sense of the word.

As an example, in this paper we apply Itacio to WaveX, a component-based system for real-time audio processing. WaveX processor modules can be combined in different topologies to achieve the desired effect. Provided that each component is described in terms of its operation restrictions and guarantees (by means of Horn clauses), the Itacio inference system will statically validate the chosen composition or pinpoint the offending connections (and explain the reasons for failure).

KEYWORDS: software components, component model, verification, constraint logic programming, software engineering.

1 Introduction

The Itacio component model [1, 2, 4, 9] is mainly a Software Engineering method for verifying the composition of software components. Its motivation arose from professional experiences in software development. It became clear that the most popular component models, such as COM, CORBA or JavaBeans solved cross-platform or low-level interoperation problems, but they did not offer the developer much protection against the violation of functional restrictions of the involved components. Many of the defects of the software could have been avoided, provided that all previous knowledge about the components was taken into account. All too frequently, the problem was not caused by a malfunction of any of the components, but by an incorrect connection of a component with its neighbours; a sort of emergent behaviour.

This contrasted with the protection a programmer enjoys regarding *signatures*. In strongly typed languages, the compiler checks all subroutine or method calls and points out every type violation in a *static* manner (at compile time), without the need of running the program. At the component level there exists a similar protection (describing interfaces with IDL or type libraries). But nothing protects the programmer beyond type and signature checking; as far as other kinds of restrictions are involved, the programmer or designer is left alone with a natural-language (and hence ambiguous) documentation, and he must guarantee that all this information has been correctly interpreted and taken into account. The correctness and completeness of this documentation does not play any direct role in automatic tests.

We felt the need of a way for:

- Expressing all that a component developer knows about the component
- Using this information in a static and automatic verification process

Also, we had some constraints in mind:

- The solution should be easily adopted by an organization, without a very specialised training
- The solution should not require a significant advance in the state of the art of any branch of computer science

The result of our research in this area is the Itacio component model. As for the knowledge expression problem, we found Constraint Logic Programming to be a convenient solution: it is very flexible, and the inference (here verification) process is clearly automatable. In addition, it can be handled by a typical developer with an affordable training (both in terms of time and cost), since it usually does not imply a mentality shock. We found that other approaches that could be of interest for verifying components did not fit well the above requirements. For instance, formal methods seem difficult to use by the average developer; even if it were false, this is the way they are perceived by the industry [14]. Other initiatives, such as OCL [8], are oriented to modelling only; OCL explicitly warns about being considered as "executable" [8, p.1]. Logic programming addresses both (and other) limitations.

Logic programming (and its advanced versions) is not a widespread practice in the development industry; our advocacy of this long-tradition technology as a companion for software components is initially surprising in peer-to-peer presentations, but after the necessary explanations we usually get positive feedback. Other authors agree that logic programming can fill significant gaps in current software component technologies [11]. Ongoing projects try to give logic programming languages the necessary features to allow them to be used as the basis of big projects, efficiency among others [17]; but we think that even using them only as an additional tool in combination with fully imperative implementations can bring significant benefits.

In this paper, the Itacio component model is briefly introduced. Then (and in a totally independent manner) the WaveX real-time sound processing system is described. Afterwards, the application of Itacio to the verification of WaveX processor designs is presented. Finally, conclusions are obtained from these experiences and future research and development lines are presented.

2 The Itacio Component Model

The Itacio component model offers a way of verifying software systems built by joining components. The main advantages of this model are that no execution of the program is needed, the specification system is fully modular and, in addition, it can be easily supported by a Constraint Logic Programming system. Finally, this model can be applied at different levels of abstraction. The notion of *component* in this model is deliberately open, so the user can apply a general verification framework to a very wide spectrum of problems beyond typical binary components [3].

A precise description of this model can be found in [2]. The central idea of the Itacio model is a flexible definition of a component. A *component* C is an entity which has a *frontier* F(C) and a set of *restrictive expressions* E(C).

F(C) is a finite set whose elements are called *connection points*; these connection points can be *sources* (whose set is denoted by S(C)) or *sinks* (whose set is denoted by K(C)). Informally stated, sources carry information outside of a component (e.g., a function call) and sinks introduce information into a component (e.g., a function's entry point). Components are considered to be *black boxes*; their only observable behaviour is described in terms of F(C) and E(C). A consequence of this is that the only possible source of errors is a bad connection between components. That is not necessarily true in the real world, but this axiom allows Itacio to become functional. If the internal behaviour of a component must be verified, the model can be applied again at a lower level of abstraction.

Restrictive expressions are also divided into two disjoint subsets. The set of *requirements* R(C) contains restrictive expressions that are Horn clauses over the sinks. The set of *guarantees* G(C) contains Horn clauses over both sinks and sources. In addition, there is a one-to-one correspondence between the sinks and the requirements (there is one requirement predicate associated to each sink, although this predicate can refer to more than one sink). Requirements do not refer to sources because, as said above, this system intends to verify the *composition* of components, not the internal behaviour of a component; so it is assumed that the component manufacturer has control over the behaviour of the component itself and he does not need to restrict its own outputs. Maybe another component will (in its restrictions over its own inputs or sinks).

A *system* $\Omega = \{\nu, \varepsilon, \boldsymbol{L}\}$ is a finite graph whose nodes ν are components and whose edges ε are source/sink pairs, together with a set $\boldsymbol{L}$ of auxiliary predicates called the *library*. Thus, a system is built by taking components and connecting each and every source with some sink, and adding some auxiliary predicates. The first requirement for a system (the so-called *topological correctness*) establishes that there will be no isolated connection points (although this is a possible extension for the system; see future work at the end of this article).

The *knowledge base* for the system, $\boldsymbol{K}(\Omega)$, is built by following an iterative substitution process over all the source and sink names so that, if some $s_i \in S(C_m)$ and $k_j \in K(C_n)$ are connected, a new, unique atom name a is generated and a new version of the involved rules (be it requirements or guarantees) is generated as needed, substituting all the occurrences of s_i and k_j by a in these generated rules. The $\boldsymbol{K}(\Omega)$ resulting from this process implicitly contains the information about the topology of the system and all the deducible rules that relate inputs and outputs of the components. The building process also ensures that the relationship between each resulting requirement and its associated sink is not lost.

Finally, the verification model relies on an inference process over $\boldsymbol{K}(\Omega)$. The system is considered to be correct if each and every requirement of $\boldsymbol{K}(\Omega)$ is proven to be true. Also, since each requirement in $\boldsymbol{K}(\Omega)$ is related to one sink, if some requirement is not fulfilled it is possible to know exactly which connection point is failing and why.

A first prototype for this model was implemented with a diagramming tool [1], which allowed to make an initial test for the basic ideas; after that, a Java/XML/VML prototype (with a web-based user interface) was built [2, 3]. Experiments with this second prototype allowed to refine the structure of the model and to test its application to new abstraction layers of the development process to verify that this simple schema is useful and will behave as expected in different situations. It was applied to time evolution of reuse contracts [3], to remote personal computer diagnostics [5] and others. Although none of this use cases had been planned when the model was described, it was successfully applied with no modifications. This gave confidence on its generality.

Currently, a third prototype is in its final development stages. The user interface is web-based (making use of ASP, XML and VML), the information about components and systems is stored in a database (previous versions used text files), and the inference engine is the ECLiPSe CLP System [6], as it had been also in previous versions. This third version of Itacio has been used for the experiments that led to this paper. Although this is a prototype and its usability could be clearly improved, it must be noticed that the technologies that would be involved in that improvement are widely available and well-known.

3 The WaveX Sound Processing System

WaveX is a real-time sound processing system developed in C++, making use of the Microsoft Visual C++ 6.0 compiler. The Microsoft Foundation Classes, or MFC [15], an object-oriented framework included with the compiler, are also used, specially for the GUI.

The goal of WaveX is to bring an audio processing system that leverages the processing power of modern personal computers. Professional, specific sound processing devices are costly, whereas current personal computers have reached a computing capacity that enables software to be used as a real-time digital sound processor. In addition, sound capture and play devices (i.e., sound cards) are widespread and affordable; in fact, it is not unusual that modern motherboards include them onboard. There are already sound processing systems which take advantage of personal computers, although they frequently advocate the use of additional hardware [10, 18]. Many other products are available for audio editing, but they are not usually oriented to real-time processing [7].

Hence, the use of domestic, general-purpose computing equipment in place of specific sound processing devices becomes a cheap and convenient alternative for many users. Some milliseconds of sound are digitised (by means of an Analog-to-Digital Converter) and stored in a buffer of discrete values that represent the amplitude of the signal at regular time lapses; the processing can be done numerically over these values, and the Digital-to-Analog hardware converter generates the resulting analog wave that we can hear [16]. Modern personal computers offer high processing capabilities which allow these computations to be fast enough to provide a real-time source of sound; all that is needed is to repeat the same operations on the next buffer while the previous one is being played by the sound hardware, without any interruption.

In general, sound processing is done by combining different stages, i.e., devices whose inputs and outputs are connected by wires (like distortion pedals, mixers, and the like). Since WaveX intends to substitute this structure by software, it seemed a good candidate for component-based development. A general framework was defined, and then a module was implemented for each desired effect (this process continues nowadays). The user of WaveX can describe the so-called *topology* of the sound processor in a description text file (more details on this later), and when this file is loaded by WaveX it sets up all the necessary components and their connections. Of course, the huge flexibility of software is a great advantage, since new modules for specific effects can be developed and used in the system at very low cost.

We decided to build WaveX as a Microsoft Windows application, where the modules would be Dynamic Link Libraries (DLLs) with a defined interface. A proprietary interface schema was defined for these DLLs. No middleware (such as COM or CORBA) was used for several reasons. First, efficiency and development simplicity were important factors. Thus, the time overhead of middleware calls was undesirable; so was the development complexity overhead involved. Also, middleware would not have brought any remarkable benefit in this case. No inter-process or inter-machine communication / deployment was planned, no inter-application communication seemed necessary, no multiple language support was desired.

The initial version of WaveX includes several components. The example presented here is deliberately simple, since the focus of this paper is not the sound processing system itself; thus, only some of the possible components will be described here for simplicity and space reasons –enough to get a basic understanding of what the system is and does.

DV_WaveInDevice captures the sound being digitised by the sound card and supplies it on its only output.

DV_WaveOutDevice receives a sound stream on its only input and plays it on the PC hardware.

DV_WaveGeneratorDevice generates a sound with certain features and supplies it in its only output (it is used mainly for testing purposes).

EF_Compression: Compression effect. It receives a sound stream on its input, and raises the amplitude of the weaker signals whereas the stronger signals are less affected. The dynamic range of a signal describes the range of loudness from the quietest signal in a recording to the loudest one; the result of compression, thus, is a smaller dynamic range, something that can be needed because the recording device can register a limited dynamic range (so without compression some sounds would be lost) or because of personal preferences.

EF_Distortion: The amplitude of a signal can be limited to certain maximum levels; if the original wave goes below or over the limits (saturation), it is cut off. In the real world it can happen because of circuitry or device recording practical limitations (and it is usually an undesired effect), but it is also deliberately used in certain cases (electric guitars are often distorted with specific devices).

EF_Echo: Echo results from taking a signal and adding the same signal with certain time delay and possibly with a lower amplitude. This component implements several kinds of echo effects: conventional echo adds the displaced signal indefinitely but with a progressive decay, with the effect of an ever quieter repetition (just as natural echo). Delay adds the signal only once; the effect is the same sound played twice but not simultaneously (the delay is usually very short). Reverberation tries to emulate the effect of the sound echoing from different walls (at different distances and angles) in a room, and this effect is achieved by adding the signal with different delays and decays. EF_Echo is a sample of a component whose behaviour can be highly parameterised.

EF_Gain: This component simply multiplies the amplitude of the signal by a factor. A factor of 2 produces a signal which is “twice as louder” as the original one; a factor of 0 produces silence.

EF_SepChannels: This component has one input and two outputs. The input signal is supposed to be stereo, and the left and right components (channels) are separated into two mono signals.

EF_JoinChannels: This component has two inputs and one output. It receives two mono sound streams, and combines them into an stereo output signal.

```
MODULE Input DV_WaveInDevice.DLL
     PARAM DesiredChannels 2
     PARAM DesiredBitsPerSample 16
     PARAM DesiredBufferSize 4096
     PARAM DesiredSamplesPerSecond 44100

MODULE Separation EF_SepChannels.DLL

MODULE Joining EF_JoinChannels.DLL

MODULE Play DV_WaveOutDevice.DLL
     PARAM DesiredChannels 2
     PARAM DesiredBitsPerSample 16
     PARAM DesiredSamplesPerSecond 44100

LINK Input out Separation in
LINK Separation left Joining right
LINK Separation right Joining left
LINK Joining out Play in
```

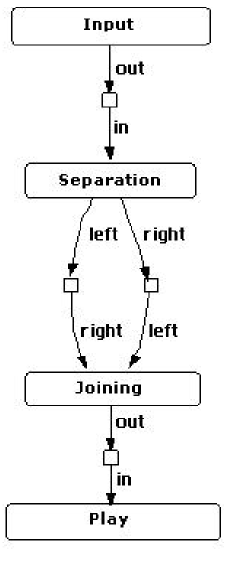


Fig. 1. A WaveX script which describes a system for inverting stereo channels; on the right, graphical representation of this system.

Many other components can be (and are being) implemented: noise gates, frequency filters, mixers, etc. Also, WaveX will be extended to process signals from disk and to record them to disk. All that is needed is to create new components. The core of the system will also be extended to synchronize different signals, so that it can be used as a small, cheap recording and mixing studio. For more information on the WaveX project, see [19].

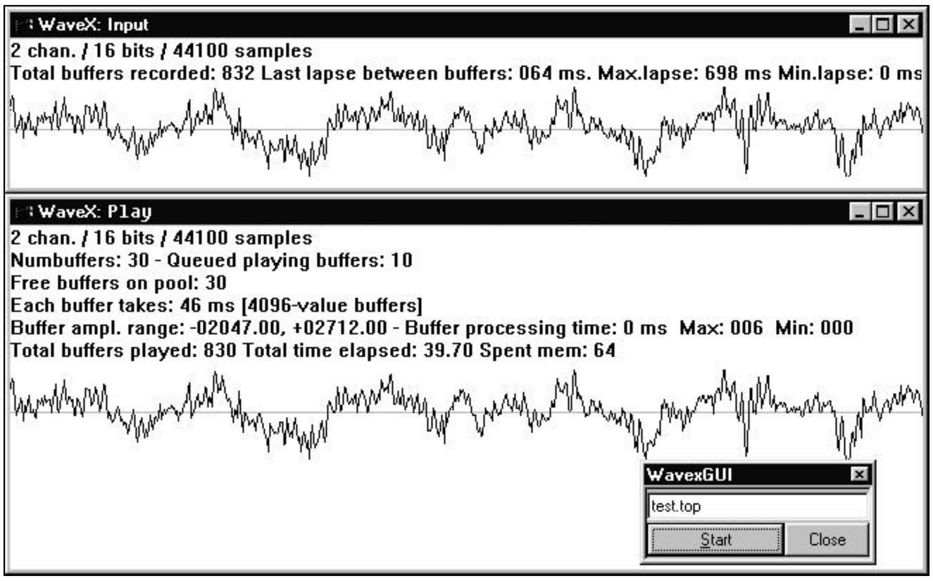
As said above, these WaveX components have well-defined inputs and outputs. Their operation can also be modified by means of different parameters. For instance, DV_WaveInDevice can be set up to record signals at different sampling rates (8000, 11025, 22050 and 44100 samples per second), different resolutions (8 or 16 bits per sample), different number of channels (1 for mono signals and 2 for stereo signals), and different buffer lengths. DV_Gain can raise or reduce the amplitude of a signal depending on the gain factor.

There is a tension between parameters and sinks; the role of parameters is to allow the creation of component instances with some degree of freedom over its behaviour, without the need of creating additional sinks and “constant” components just to represent the parameter values. This would add unneeded complexity to the design.

As said above, the user can write a “topology” file that describes certain configuration of components. For instance, the script in Fig. 1 inverts left and right channels of a stereo signal. The involved keywords are very simple: the `MODULE` statement declares a module instance, indicating which component (DLL) implements it. The `PARAM` statements follow the component they affect. The `LINK` statements refer to the declared component instances, giving information on how they must be connected. The syntax can be easily deduced from this sample:

```
MODULE <Name> <ComponentDLL>
PARAM <ParamName> <Value>
LINK <OriginModule> <Source> <EndModule> <Sink>
```

The WaveX core loads and interprets this script, creating the working system by combining the necessary components (see Fig. 2).



```
2 chan. / 16 bits / 44100 samples
Total buffers recorded: 832 Last lapse between buffers: 064 ms. Max.lapse: 698 ms Min.lapse: 0 ms
```

```
2 chan. / 16 bits / 44100 samples
Numbuffers: 30 - Queued playing buffers: 10
Free buffers on pool: 30
Each buffer takes: 46 ms [4096-value buffers]
Buffer ampl. range: -02047.00, +02712.00 - Buffer processing time: 0 ms  Max: 006  Min: 000
Total buffers played: 830 Total time elapsed: 39.70 Spent mem: 64
```

Fig. 2. The sample WaveX system in action: the core (WavexGUI), the Input component and the Play component. The Separation and Joining components have no visual representation.

With this simple example, it seems that no verification is needed. But even in this case, errors can be made. For instance, if the Input device is configured to record a mono signal (`PARAM DesiredChannels 1`) the system will not work. The Separation component needs a stereo signal, whereas Input would be configured to produce a mono signal. A special difficulty is that problems could arise at distant components; for instance, if Input was configured to record at 44100 samples per second but Play was configured to play at 22050 samples per second, the problem would manifest at the connection between Play and Joining (there is really no problem until that point, since the intermediate components can handle any sampling rate). Some devices may require a limited amplitude margin (for instance, certain playing hardware may even be damaged by a too loud signal, so it could be interesting to require limits somewhere). Of course, WaveX is designed to support much more complicated systems (involving more components and more interrelated parameters) than the small example of Fig. 1, and there are lots of potential malformations.

There is still an important problem when WaveX is used in real time. As said above, each buffer holds a certain lapse of the sound; its duration depends on buffer size, sampling rate and the quality (resolution and number of channels) of the signal. The buffer generated by a recording component is passed from one component to another, being processed in different ways, and usually ends in a playing component. The time window available for doing this is the duration of a buffer; if the time needed for fully processing a buffer is longer than the lapse of sound a buffer holds, the system will not be prepared for immediately processing the next recorded buffer, since the recorder produces buffers at a constant and uninterrupted pace.

Of course, it is possible to try to tackle these connection problems with "traditional" pre/postconditions or assertions [12]. But this has some disadvantages:

– The interface description (and the knowledge about the intended use) of a component will be buried in the processing source code, mixed with it.
– The process of handling a mismatch does not end simply detecting it; the error condition must be reported, described and properly handled, and coherent exception handling may not be an easy task (especially when separate components are involved, as in this case).
– Assertion-based verification shows mismatches only if assertions are violated during execution; to a certain point, they are equivalent to testing.
– In general (at least in widespread development tools and in most programmer's habits) imperative language assertions are not statically analysed. They must be run; the system must be really built and tested, instead of verifying the design in advance.

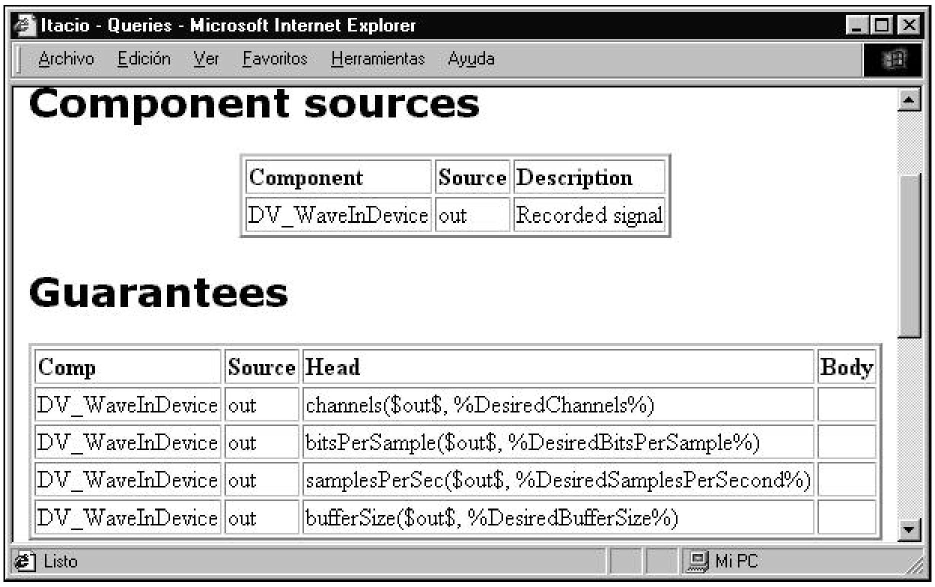
4 Applying the Itacio prototype to WaveX

Provided that WaveX is a very versatile sound processor, and its user is going to combine multiple off-the-shelf components in many different ways, we can expect that construction to be a quite error-prone task; in addition, the final version of WaveX will have many more components, and maybe some of them will work only with certain kinds of signals, so the user will need also to include adaptors in his design, raising complexity. It is desirable to help the designer detect potential problems as soon as possible (at design time); moreover, a system that can directly point out the inconsistencies and explain them would be of value.

Thus, Itacio comes into scene. Instead of cluttering the different components with C++ verifications (and handling the different error conditions), Itacio can be used to describe the WaveX components and verify each design before really building or running the system. This process will be described now. It must be noticed that, unless explicitly stated, the Itacio prototype used with WaveX is exactly the original one, and no special abilities have been added for WaveX.

Before any use of Itacio can be done, we must make an **instantiation** of the Itacio model (described in a previous chapter) defining the concepts of component, source, sink, etc. in the target domain. In this case, it seems obvious that each WaveX component, with its inputs and outputs, can be directly identified as an Itacio component with its sinks and sources, respectively.

With this strategy in mind, the first task is to insert in Itacio the structural definition of all the possible components: their name, sources and sinks. Once this has been done, we can use logic programming statements to write down the requirements and guarantees of each component separately. As an example, Fig. 3 shows the



Component sources

Component	Source	Description
DV_WaveInDevice	out	Recorded signal

Guarantees

Comp	Source	Head	Body
DV_WaveInDevice	out	channels(out, %DesiredChannels%)	
DV_WaveInDevice	out	bitsPerSample(out, %DesiredBitsPerSample%)	
DV_WaveInDevice	out	samplesPerSec(out, %DesiredSamplesPerSecond%)	
DV_WaveInDevice	out	bufferSize(out, %DesiredBufferSize%)	

Fig. 3. Frontier of DV_WaveInDevice (Itacio prototype screen capture)

frontier of the DV_WaveInDevice component; this definition is a general template for that component, and instances of it will be created in each system as needed. In this case, the component does not have any sink. It has only one source (called *out*), and it offers (at the moment of the screen capture) four guarantees.

It can be seen that guarantees are in fact Prolog code under the form of Horn clauses; in this case, none of the rules has a body, so they are all facts. The Prolog code adheres to a couple of conventions: the atom names enclosed by “$” must refer to sources or sinks of that component, and they will be automatically substituted by the atoms that represent the corresponding connections (as described earlier). The atom names enclosed by “%” represent parameters, and they will also be substituted by the specific values of each component instance. In this case, the DV_WaveInDevice component guarantees that the signal it produces will have the required number of channels, resolution, sampling rate and buffer size.

Analogous information would be entered in Itacio for all the components that could be used to build a system (in this case, specific sound processors). A second example, the description of EF_SepChannels, is shown in Fig. 4 (represented in text form and not as a screen capture for space reasons). It can be seen that this component requires its input signal to be stereo, and compromises itself to generate a mono left signal in which all other features remain unchanged; the same for the right one.

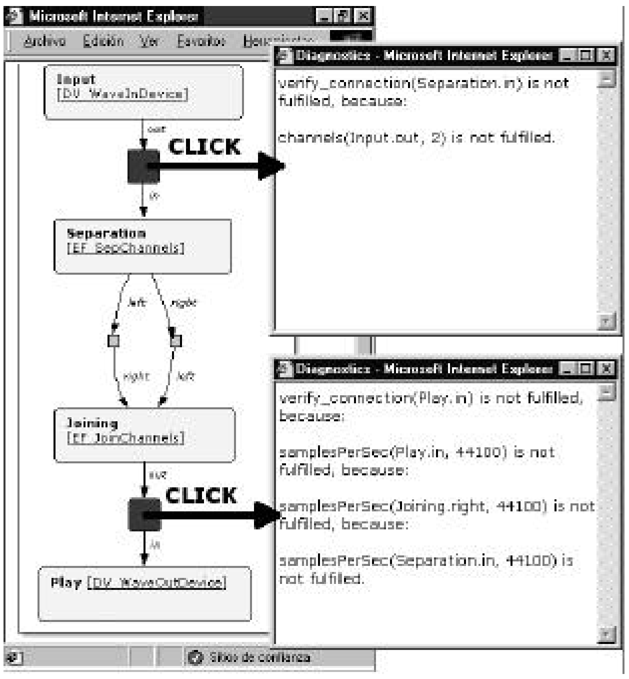
Component: EF_SepChannels	
Sinks: in	**Sources**: left, right

Requirements:
```
        channels($in$, 2).
```
Guarantees:
```
        channels($left$, 1).
        bitsPerSample($left$, X) :- bitsPerSample($in$, X).
        samplesPerSec($left$, X) :- samplesPerSec($in$, X).
        bufferSize($left$, X) :- bufferSize($in$, X).

        channels($right$, 1).
        bitsPerSample($right$, X) :- bitsPerSample($in$, X).
        samplesPerSec($right$, X) :- samplesPerSec($in$, X).
        bufferSize($right$, X) :- bufferSize($in$, X).
```

Fig. 5. Frontier of EF_SepChannels.

Having the template components defined, the final steps would be the creation of a system. We only need to define instances of the existing components and connect their sources and sinks as needed; also, we need to give the desired values to the parameters of each component instance. For this task, the current version of Itacio offers a poor, database-style interface, but of course a good and easy to use graphical editor could be developed.

Following this procedure, we can have the


Microsoft Internet Explorer
Archivo Edición Ver Favoritos Herramientas
Input [DV_WaveInDevice]
out
CLICK
in
Separation [EF_SepChannels]
left
right
right
left
Joining [EF_JoinChannels]
out
CLICK
in
Play [DV_WaveOutDevice]
Sitios de confianza
Diagnostics - Microsoft Internet Explorer
Diagnostics - Microsoft Internet Explorer

verify_connection(Separation.in) is not fulfilled, because:

channels(Input.out, 2) is not fulfilled.

verify_connection(Play.in) is not fulfilled, because:

samplesPerSec(Play.in, 44100) is not fulfilled, because:

samplesPerSec(Joining.right, 44100) is not fulfilled, because:

samplesPerSec(Separation.in, 44100) is not fulfilled.

Fig. 4. A system with errors and their explanations.

sample system defined. Although Itacio does not offer graphical *editing* facilities yet, it is capable of *showing* a graphical representation of any system, in Web format (by means of XML / VML [13]) and in PostScript format. Besides the usefulness of this depiction for a better understanding of the system, the most interesting aspect of it is that it also incorporates the results of the verification process.

Itacio can generate the Knowledge Base that represents a system (as described earlier in this paper) and interact with the ECLiPSe inference engine to obtain information about the correctness of each of the connections. If some connection is not correct, the graph will clearly show it; the user can then click on the offending connection and get an explanation of the failure. Fig. 5 represents the case that the Input component has been configured to generate a 22050 Hz mono signal whereas the Play component expects a 44100 Hz signal and the Separation component needs a stereo signal. The system pinpoints the offending connections with a big dark square; when the user clicks on that square, explanations about the problem are offered so that he can correct his design.

As for the real-time processing problem, we can easily add the guarantees and restrictions for taking this issue into account. Simply, the DV_WaveOutDevice component will expand its requirements to add the following terms:

```
   buffer_milliseconds($in$, BUFFER_TIME),
   buffer_processing_time($in$, PROCESSING_TIME),
   PROCESSING_TIME < BUFFER_TIME
```

As we can see, an auxiliary predicate is needed in order to calculate the time a buffer holds, and this is not bound to any particular component. It is general knowledge, so it will be included in the system library with a simple rule:

```
buffer_milliseconds(SIGNAL, TIME) :- channels(SIGNAL, CHANNELS),
   samplesPerSec(SIGNAL, SAMPLES_PER_SECOND),
   bufferSize(SIGNAL, BUFSIZE),
   TIME is ((1000 * BUFSIZE) / CHANNELS) / SAMPLES_PER_SECOND.
```

Also, each component would incorporate its own information about performance. In this case, we have included only worst-case processing time in milliseconds, and this information can be obtained empirically or be computed from algorithm complexity. In this case, EF_SepChannels could have a worst-case processing time of about 10 milliseconds:

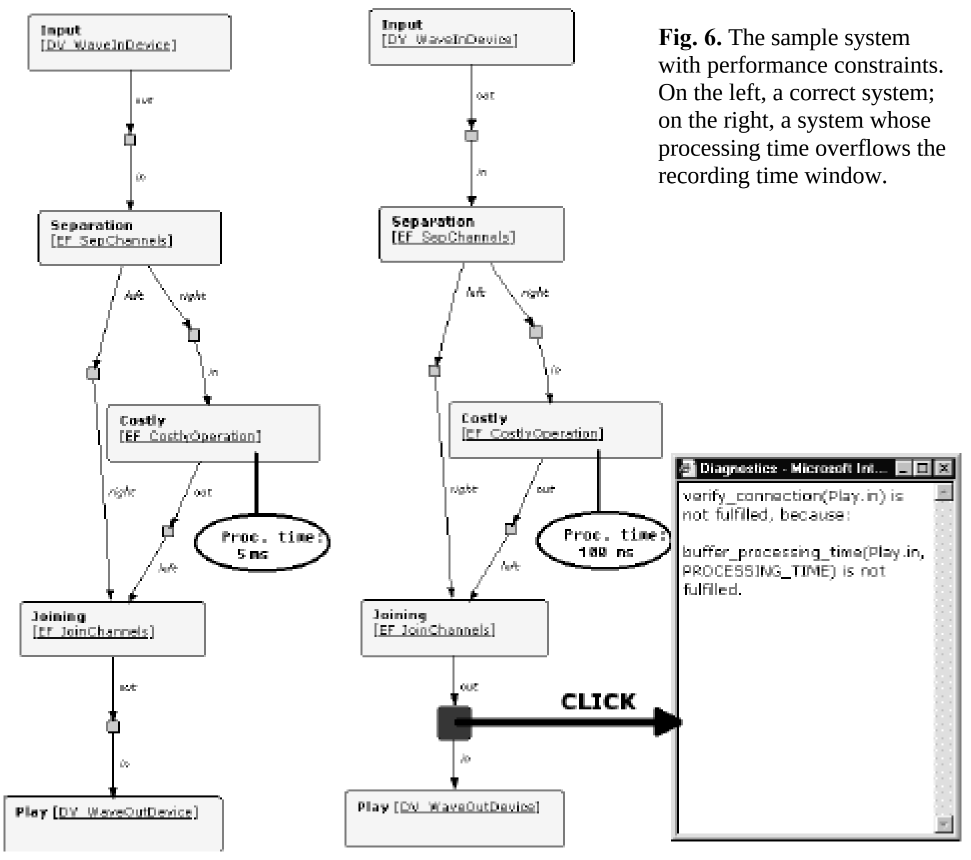
```
buffer_processing_time($left$, X) :- buffer_processing_time($in$,
   TIME_INPUT), X is 10 + TIME_INPUT.

buffer_processing_time($right$, X) :- buffer_processing_time($in$,
   TIME_INPUT), X is 10 + TIME_INPUT.
```

In the case of EF_JoinChannels, provided that its own processing time is always under 10 ms, the behaviour of this component would force it to wait for its two inputs (left and right) so the accumulated processing time would be the worst of both adding its own 10 ms:

```
buffer_processing_time($out$, X) :- buffer_processing_time($left$,
   LEFT), buffer_processing_time($right$, RIGHT), LEFT > RIGHT,
   X is LEFT + 10.
buffer_processing_time($out$, X) buffer_processing_time($left$, LEFT),
   buffer_processing_time($right$, RIGHT), LEFT =< RIGHT,
   X is RIGHT + 10.
```

It must be noticed that, since this example is deliberately simple, no constraints are involved in the declarative description; we are using only worst-case processing times, and a simple comparison is enough to verify that the time window is not surpassed. But if a deeper analysis of the system behaviour was needed, we could use processing time ranges, which would require a CLP system since traditional Prolog is not well suited to unify or compare domains.

Introducing these requirements and guarantees in Itacio, the result is a more detailed description of the behaviour of the components. In this case, the system is still valid, because the processing time clearly fits in the time window available; a buffer of 8192 samples for a stereo, 44100 samples per second signal can hold 92 ms,



```
verify_connection(Play.in) is
not fulfilled, because:

buffer_processing_time(Play.in,
PROCESSING_TIME) is not
fulfilled.
```

Fig. 6. The sample system with performance constraints. On the left, a correct system; on the right, a system whose processing time overflows the recording time window.

whereas processing time in the worst case is about 20 ms with the performance data we have entered (and these data were overly pessimistic) . But if we added a more complex component, things could change. As an example, we have prepared a component for testing purposes, which simply fakes a long processing time. If we introduce this component as an additional “transformation” between the right output of EF_Separation and the left input of EF_Joining, nothing happens if we configure it to spend 5 ms of time; but if we configure it to spend 100 ms, the time window is overflowed, and the system detects this as usual (Fig. 6).

A final comment must be done about the development process for WaveX modules. After the suitability of Itacio for verifying WaveX designs was established, a specific functionality was added to Itacio: the ability to generate the WaveX topology description text file from the system description stored in Itacio. This is a very simple database query and translation process, but it allows the WaveX user to use Itacio as a semi-graphical design environment; once the design is correct and validated against all the available knowledge about the involved components, the user can directly see the system working. This tight integration between the verification tool (Itacio) and the development/assembling tool (WaveX in this case), and a good implementation of the verification tool (with convenient user interfaces and graphical editors) is probably the key to effectively incorporate Itacio into any development process.

5 Conclusions and Future Work

The signature verification mechanism that component models such as COM, CORBA or JavaBeans offer today seems unable to protect developers and users against incorrect component combinations. We believe that component behaviour should be described in a manner that allows all the knowledge about a component to be used in an automatic, static verification process. We also believe that the proposed method should be feasible in the sense that it can be learnt by average programmers and that it should not require a huge investment in tools. Among the different specification tools available, logic programming seems to be a very good alternative, because it allows the declarative knowledge to be collected and used in an automatic inference process, and it is flexible enough to express very different kinds of restrictions.

The Itacio component model was designed with these motivations in mind. It has been applied at different abstraction levels with success, and the present paper describes its application to a component-based real-time sound processing system, called WaveX. Although the samples presented here are intentionally simple for space reasons, they show that the generality of logic programming is a great benefit for describing all kinds of component use restrictions that could not be checked with current commercial component platforms.

It can be argued that the Itacio approach forces the developer to learn a “second language”. It is indeed true that collecting and using knowledge in the design / development process does not come for free; but we believe that the Itacio proposal is clearly feasible. Any other specification technique, including many formal methods, would also require an extra effort for it to be effectively used. Itacio is based on (declarative) programming and on a very well known inference process, supported by available -or easy to develop- tools, and these factors may be of value for its adoption in a development organization.

Although the WaveX system is not the focus of this paper, we expect it to grow with much more components, and we expect the Itacio component model to play an important role to help the final user in the design of sound processors. In this case, the general Itacio prototype has been used, but in future versions of WaveX, the verification system could be *embedded* in the final product, greatly improving user experience.

Future research efforts could approach several problems:

- The elimination of the closed component graph requirement (what was defined in Itacio as *topological correctness*). This would allow to verify systems in which not all sources and sinks are connected; in other words, this would allow to verify unfinished systems.
- A more intelligent diagnostic process which extracts all useful information from the knowledge base that describes a system, giving better information on errors or even suggesting useful components for finishing a system.
- The development of a full-fledged Itacio-based verification system (only prototypes have been developed so far) with graphical editors, a good user interface and a tight integration with other development tools.

References

1. Cernuda, A., Labra, J. E., Cueva, J. M. *Itacio: A Component Model for Verifying Software at Construction Time*. III ICSE Workshop on CBSE. 5-6 June 2000, Limerick, Ireland. `http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html`
2. Cernuda, A., Labra, J. E., Cueva, J. M. *A Model for Integrating Knowledge into Component-Based Software Development*. KM - SOCO 2001, 26-29 June 2001, Paisley (Glasgow, Scotland). ICSC Academic Press, ISBN: 3-906454-27-4.
3. Cernuda, A., Labra, J. E., Cueva, J. M. *Verifying Reuse Contracts with a Component Model.* 6th JISBD, 21-23 November 2001, Almagro (Spain).
4. Cernuda, A., Labra, J. E., Cueva, J. M. *Verificación y validación mediante un modelo de componentes*. SISOFT-2001 - Bogotá (Colombia), 29-31 August 2001 (Spanish only). `http://atenea.udistrital.edu.co/eventos/simposio/`
5. Cernuda, A. *Diagnóstico remoto de configuración de componentes software en Windows (Remote Diagnostics of Software Components Configuration under Windows).* Poster at the University of Oviedo Technology Transfer Conference, 13-14 September 2001 (Spanish only). `http://www.uniovi.es/Vicerrectorados/Investigacion/portal/ot/activos/Diagnóstico%20remoto.PDF`
6. ECLiPSe web site. `http://www.icparc.ic.ac.uk/eclipse`
7. GoldWave audio editor, `http://www.goldwave.com`
8. IBM and others. *Object Constraint Language Specification, version 1.1*. September, 1997.
9. Itacio project web page. `http://www.agustincernuda.com/itacio_eng.html`
10. Kyma (a product by Symbolic Sound Corporation), `http://www.symbolicsound.com`
11. Lau, K. K. *The Role of Logic Programming in Next-generation Component-based Software Development*. Proceedings of Workshop on Logic Programming and Software Enginering, London, July 2000 (edited by Gupta, G. and Ramakrishnan, I. V.)
12. Meyer, B. *Object-Oriented Software Construction*. Prentice Hall, 1988. Second edition: ISBN 0136291554, April 1997.
13. Microsoft VML Overview. `http://msdn.microsoft.com/workshop/author/vml`
14. Pressman, Roger S. *Software Engineering*. McGraw-Hill, 1992.
15. Prosise, Jeff. *Programming Windows 95 with MFC*. Microsoft Press.

16. Smith, S. W. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997. ISBN 0-9660176-3-3. Available online at `http://www.dspguide.com/pdfbook.htm`
17. Somogyi, Z., Hendersgon, F. J. and Conway, T. C. *The implementation of Mercury, an efficient purely declarative logic programming language*. Proceedings of the ILPS '94 Postconference Workshop on Implementation Techniques for Logic Programming Languages. Syracuse (New York), November 1994.
18. Waves (KS Waves, Ltd.) `http://www.waves.com`
19. WaveX project web page. http://www.agustincernuda.com/wavex_eng.html