Constraints Derivation and Propagation for Large-Scale Embedded Systems Exploration

Laurenţiu Nicolae[1], Ed Deprettere[1]
[1] Leiden Institute for Advanced Computer Science
Niels Bohrweg 1, 2333 CA Leiden, Netherlands
Email : {nld,edd}@liacs.nl

Abstract— **The translation of user requirements to system constraints and parameters during an exploration exercise is a hard problem, especially in the context of large scale embedded systems. This process is almost never simple and straightforward, and it often requires multidisciplinary skills. The user requirements are not fixed during the exploration stage. Furthermore, the system constraints may vary according to the design choices made in the course of the exploration and development process.**

In this context, a need for hot-linking the constraints of the architecture to the top-level requirements of the application becomes apparent. We consider the system model as separate hierarchies of application and architecture components, coupled by a mapping layer. We aim towards an automatic derivation of these constraints downwards to all the levels of the hierarchy. Furthermore, we employ a procedure of validation and adjustment of these constraints in the lowest levels, to eliminate possible inconsistencies.

I. Introduction

The design of a large scale system is usually multidisciplinary, and a great deal of effort is invested into translating abstract user requirements into hard numbers that can be used in the design cycle. These requirements become thus design objectives, such as the need for a nominal throughput, latency, power consumption, cost etc. The objectives are usually expressed in terms of constraints associated with the system building blocks. This is usually done by experts that evaluate different concepts and baseline systems at a very high level of abstraction.

Once one or more system concepts are selected, they are taken through a system-level exploration. This stage takes as an input the translated top-level requirements and baseline systems and tries to evaluate different design decisions while remaining at a reasonably high level of abstraction. If critical components are detected in this stage, they can be taken out of the system and evaluated separately in a component-level exploration exercise [1]. This exploration process is illustrated in figure 1.

The paradigm we are employing when designing large scale digital signal processing systems requires the partitioning of the system into three views: application, architecture and mapping. We use this separation of concerns as a way to master the complexity inherent to such systems. Each of these views is further decomposed into hierarchies of interconnected components. Our goal is to compare different implementation concepts for a set of given requirements and answer "what-if" questions at the system level as early in the design cycle as possible.

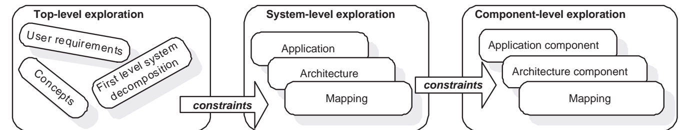

Fig. 1. Exploration can be performed at different levels of abstraction

A system exploration exercise consists of several stages. The first stage is dealing with the qualitative or functional requirements and constraints dictated by the top-level, such as the choice of algorithms, the signal quality etc. It first establishes a fixed set of top-level requirements. A hierarchical topology of an application, an architecture and the mapping relations between the two are then specified by the designer.

The second stage introduces the quantitative requirements and constraints, such as the number of bits used for the digitization of the signal, the required throughput of the system, the maximum power dissipation and so on. The result of the second stage is an executable specification of the system which fulfills the top-level requirements, both quantitative and qualitative. This specification can be evaluated and compared with previous results. A more detailed description of the methodology is given in [2].

Ideally, the constraints of the system would be orthogonal. However, for real designs this is hardly the case. The constraints of the application are related via the mapping step to the constraints of the architecture. Thus the modifications of the model in one domain are related to the choices made in the other domain in a two way traceable manner. For instance, if a complex filtering algorithm has been selected as an application and a new quantitative requirement for this algorithm is a very high throughput, the designer may choose to select a different algorithm with a different set of parameters. Another alternative is to adjust

the architecture, by either adding more filtering blocks or by selecting a different component from the library.

When the model structure and user requirements are fixed, the system constraints are seldom modified, if at all, and this is the reason why in a normal development cycle the managing and automatic derivation of constraints is not a priority. However, this changes dramatically when one has to deal with evaluating different system realizations under alternative user requirements, which leads to ever changing constraints on the model. We argue that an automatic constraint derivation system is a crucial part of a successful exploration framework, as it allows for fast design space exploration by minimizing the amount of user interaction. Furthermore, our approach facilitates a tracing of the steps taken from requirements to specification over the whole project life cycle.

In this paper we are describing the methods and tools we apply for the manipulation of constraints for large scale systems exploration. The mathematical constraint derivation model and the constraint propagation methods are described in detail. We present a constraint validation scheme we use to assert and adjust the system constraints. Then we conclude with a real-life example of constraints derivation.

II. Motivation and Related Work

The issues of design space exploration and multi-objective optimization problem have been addressed in the past [3], but the methods employed are mainly targeted towards the optimization of smaller systems. Furthermore, they are successful in an fixed and well organized design space. Our goal is to provide methods and tools that help a designer to explore alternatives not only in terms of design choices, but also in terms of requirements.

The orthogonalization of concerns by separating the system model into multiple views has been successfully employed in other tools, such as Polis[4] for System-on-a-Chip designs. For larger, more heterogeneous systems, assigning constraints across models for verification becomes an issue that is addressed in the successor of Polis, namely the Metropolis[5] project. The model they employ is designed for co-simulation of the application and the architecture, and consists of a formal specification of the application behavior connected to the architecture layer via executable traces, together with a formal model for performance constraints.

It is indeed possible to derive the system constraints from a detailed behavior specification; however, creating this specification is both tedious and time-consuming. Therefore we argue that this approach is less suitable for high level design space exploration. Our goal is to quickly explore several alternatives and gain a certain degree of confidence before delving into lower-level modeling and final implementation.

Since we are examining different, and sometime even unrelated solutions, we need to be able to rapidly specify the system alternatives we want to explore. Therefore this exploration framework is currently based on a library of system components. The performance evaluation of the system is obtained by using stochastic models[6], also based on information taken from the library.

We propose here a formalization of the constraints model which we use to automate the derivation of constraints from user requirements. This supports our rapid specification approach, and allows us to trace the requirements across the system model.

III. Constraints Derivation

The main role of the constraints of a given system is to express quantitatively the user requirements set for the system in all the system components. Usually the constraints are obtained from the user requirements by quantification, refinement and decomposition over the system, as shown in figure 2. The system is considered a viable design solution if and only if all the constraints are met.

Let us consider again our filtering example. Given a number of sensors and a sampling rate as top-level requirements, they are translated into a throughput for the filter block. This throughput is then distributed into I/O rates for all the filter processes on the lowest level.

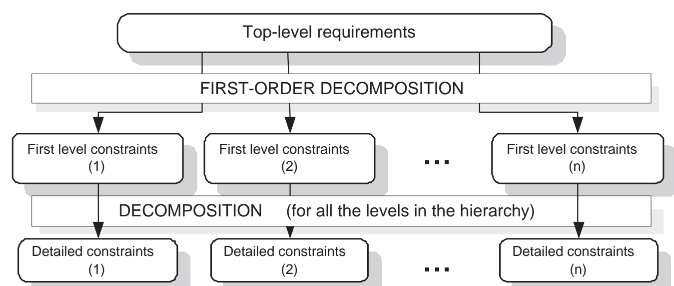

Fig. 2. Converting user requirements to system constraints

In the course of an exploration exercise, the goal is to evaluate system alternatives as early in the design process as possible. The best of them are then selected and studied in-depth. Another goal is to evaluate if an alternative is flexible enough to easily accommodate more demanding requirements. This is especially important for large scale infrastructures, which are deployed and operated for long periods of time.

A. Constraints Dependencies

In this context, the need for an automatic derivation of constraints from requirements becomes apparent. The first step in creating such a system is identifying the dependencies of the constraints at both local and system level. After careful consideration, we concluded that the constraints have the following dependencies, as shown in figure 3:

Fig. 3. System constraints do not depend only on user requirements

- *The user requirements* are obviously the main driver for the constraint derivation. However, for automatic derivation we may only use quantifiable requirements. For instance "we want a fast system" is a requirement that may be quantified by a throughput constraint.
- *The system topology*, which includes by definition the partitioning and connections of our system, is playing an important role in constraint refinement. If, for instance, we choose to split one of the components in two smaller pieces that may work in parallel, the constraints for the original component will be redistributed accordingly.
- *The instantiations* of various system components are defining parameters that may also influence constraints. If we have an algorithm that requires a certain amount of data to start, such as a matrix inversion, this requirement is translated to a input constraint.
- Various other *local parameters* that do not fit in the above categories may be defined at any level in the system hierarchy.

If we want to capture the system constraints and to derive them in an automatic fashion, all these parameters must be encompassed in a formal constraint model.

B. The Mathematical Constraint Model

The formalism that we employ to describe the system constraints is based on the quantifiable user requirements obtained after the first level translation, as shown in figure 4. A general constraint of a system component C is a condition imposed on its behavior, performance or physical characteristics. In this paper a quantifiable constraint is called constraint for the sake of convenience.

Definition 1: Let K_C be the set of constraint values assigned to the system component C. Let M_C be the set of metric values derived for the component C after the performance evaluation. Let $m_x \in M_C$ a metric and $\kappa_x \in K_C$ a constraint.

By definition, the types of constraints are:

1. Equality (E): $m_x = \kappa_x$
2. Upper-bound (UB): $m_x \leq \kappa_x$
3. Lower-bound(LB): $m_x \geq \kappa_x$
4. Range (R): $m_x \in [\kappa_x(inf), \kappa_x(sup)]$

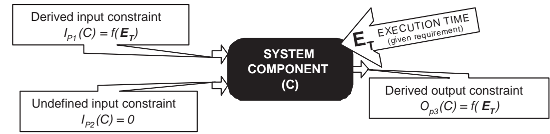

Fig. 4. A simple constraints derivation example: the I/O constraints depend on the execution time requirement of the given system component

Definition 2: Let P_C be the set of parameters that defines an instance of a system component C. Let R_C be a set of requirements associated with C. Let LP_C be the set of local parameters also associated with C.

A *constraint* of a system component C is expressed as a function $\kappa_x = f_x(r_1, ..., r_m, p_1, ..., p_n, lp_1, ..., lp_o)$, where $r_i \in R$, $p_j \in P_C$ and $lp_k \in LP_C$.

If we compare this definition with the relations expressed in figure 3, we notice that the formula does not capture explicitly the relation between a component topology and its constraints. However, we are to consider that the I/O constraints are defined on the ports of a given component, which are topological properties. Based on definition 2 we can conclude that we can define a superset of the requirements for the whole model:

Definition 3: The union of all component requirement sets for a given system model S, $\bigcup_{C \in S} R_C$, $\forall C \in S$, is called a *requirement set*, and is denoted with R_S.

The set R_S defines all the requirements needed to instantiate the constraints of a model. Therefore the definition of a "what-if" scenario is essentially reduced to the creation of a new requirement set for the current model. We can define in the same fashion a superset of the local parameters:

Definition 4: The union of all local component parameter sets for a given model, $\bigcup_{C \in S} LP_C$, $\forall C \in S$, is called a *parameter set*, and is denoted with LP_S.

Together, R_S, LP_S and the instantiation of the components in the current system completely determine the constraints. Therefore this formalism allows us to define a coherent way of deriving the constraints from the user requirements. Furthermore, by redefining R_S, LP_S or both, the designer can automatically rederive the constraints of the model, thus being able to check flexibility, scalability and system tolerance issues.

Our constraint model is is essentially a simplified subset [7]and it is mainly designed to accompany existing library

components and to provide an accessible way of specifying constraints. The simplifications derive mainly from the fact that we are evaluating the systems in their stable working state, and we neglect the transitional periods, which enables us to have static constraints that are valid during the whole life time of the system. However, we do enable the evaluation of the same system with different sets of constraints, which enable us to test the system also in border conditions and transitional states.

IV. Constraint Propagation

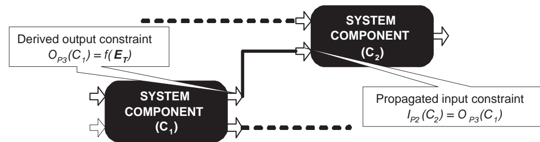

Fig. 5. The constraints are propagated over the connections between the system components

In the ideal case, after the constraint derivation we would have instantiated all the constraints in the model. However, there may be situations where the requirement set is incomplete, and some of the constraints cannot be derived. In such cases we can take advantage of the topological information encompassed in the model and use it to propagate the I/O constraints over the communication lines, as shown in figure 5.

The non-I/O related constraints may only be propagated through the levels of the hierarchy, by using aggregation rules for all the components in a given container. For instance, an aggregation rule for the processing constraints of a container Ω may be written as $P_\Omega = \sum_{C_i \in \Omega} P_{C_i} * w_i$ where w_i is a predefined weight associated with the component C_i.

If there are multiple connections on a given port, the behavior of the port may vary, and so the I/O constraint derivation will not be always consistent. That is why we have to impose certain restrictions on these topology patterns. Therefore, if an input port is writing on multiple output ports, we consider that the output is *duplicated*. If an output port reads from multiple input ports, we consider that the reading of data is simultaneous, and therefore the data streams are *summed*. The same operations will reflect upon the constraints on these ports.

The true value of constraint propagation is shown when we make use of the mapping layer to propagate constraints from the application model to the architecture model. The mapping layer describes the relations between the components in the two models, as shown in figure 6. By treating the application components as being part of the bottom-most architecture layer, we are able to apply the same

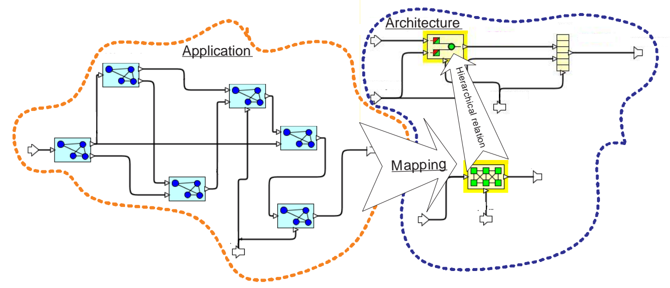

Fig. 6. The application constraints are pushed through the mapping layer into the architecture components and then further up in the architecture layer hierarchy

methods and aggregation rules as described above.

V. Constraint Validation and Adjustment

The compatibility of the derived constraints cannot be guaranteed without severely limiting the freedom of choices for the designer. Therefore we have to create a procedure that would validate the derived constraints according to the system topology, and adjust them if possible. We currently operate only on the I/O constraints, as shown in figure 7.

The example depicted in figure 7 takes into account only the case where we have two lower-bound constraints on the edges of a connection. However, there are many more cases to be considered. We are describing these cases with the use of three tables, which represent all the possible cases that we may encounter when we connect an output port O with an associated constraint κ_O to an input port I with its respective constraint κ_I.

Table I and II describe the error and warning conditions for the constraints on the two given ports. Table III shows the adjustments that may be performed automatically if a warning condition is encountered. The adjustment procedure may change one or both constraints in order to make them compatible. The procedure is reliable in most cases; however, the constraints thus obtained are flagged in the model for later review. If necessary, the designer may choose to skip the adjustment procedure and manually fix the errors and warnings that may occur.

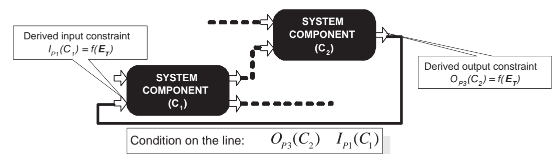

Fig. 7. If two I/O constraints are specified on a topology connection, we can verify if they are compatible

TABLE I
ERROR CONDITIONS FOR CONSTRAINTS ON A GIVEN CONNECTION

$Output/Input$	$E_I : m = a$	$UB_I : m \leq a$	$LB_I : m \geq a$	$R_I : m \in [a_1, a_2]$
$E_O : m = b$	$a \neq b$	$a < b$	$a > b$	$b \notin [a_1, a_2]$
$UB_O : m \leq b$	$a > b$	$-$	$a > b$	$a_1 > b$
$LB_O : m \geq b$	$a < b$	$a < b$	$-$	$a_2 < b$
$R_O : m \in [b_1, b_2]$	$a \notin [b_1, b_2]$	$a < b_1$	$a > b_2$	$(a_1 > b_2) \vee$ $(a_2 < b_1)$

TABLE II
WARNING CONDITIONS FOR CONSTRAINTS ON A GIVEN CONNECTION

$Output/Input$	$E_I : m = a$	$UB_I : m \leq a$	$LB_I : m \geq a$	$R_I : m \in [a_1, a_2]$
$E_O : m = b$	$-$	$-$	$-$	$-$
$UB_O : m \leq b$	$a \leq b$	$a < b$	$a \leq b$	$a_1 \leq b$
$LB_O : m \geq b$	$a \geq b$	$a > b$	$a \geq b$	$a_2 \geq b$
$R_O : m \in [b_1, b_2]$	$a \in [b_1, b_2]$	$a \in [b_1, b_2]$	$a \in [b_1, b_2]$	$(a_1 < b_1 \leq a_2 \leq b_2) \vee$ $(a_1 \leq b_1 \leq a_2 < b_2) \vee$ $(b_1 < a_1 \leq b_2 \leq a_2) \vee$ $(b_1 \leq a_1 \leq b_2 < a_2)$

TABLE III
ADJUSTMENT OF THE CONSTRAINTS ON A GIVEN CONNECTION

$Output/Input$	$E_I : m = a$	$UB_I : m \leq a$	$LB_I : m \geq a$	$R_I : m \in [a_1, a_2]$
$E_O : m = b$	$-$	$a = b$	$a = b$	$a_1 = b$ $a_2 = b$
$UB_O : m \leq b$	$b = a$	$a = \min(a, b)$ $a = \min(a, b)$	$a_1 = a, b_1 = a$ $a_2 = b, b_2 = b$	$b_1 = a_1$ $b_2 = \min(a_2, b)$ $a_2 = \min(a_2, b)$
$LB_O : m \geq b$	$b = a$	$a_1 = b, b_1 = b$ $a_2 = a, b_2 = a$	$a = \max(a, b)$ $b = \max(a, b)$	$b_1 = \max(a_1, b)$ $b_2 = a_2$ $a_1 = \max(a_1, b)$
$R_O : m \in [b_1, b_2]$	$a_1 = b_1$ $a_2 = \min(a, b_2)$ $b_2 = a$	$a_1 = \max(a, b_1)$ $a_2 = b_2$ $b_2 = \min(a, b_2)$	$a_1 = \max(a, b_1)$ $a_2 = b_2$ $b_1 = \max(a, b_1)$	$a_1 = \max(a_1, b_1)$ $a_2 = \min(a_2, b_2)$ $b_1 = \max(a_1, b_1)$ $b_2 = \min(a_2, b_2)$

VI. IMPLEMENTATION

We have implemented the constraint derivation and validation procedures in the *Massive Exploration Tool*[8], a framework that facilitates design space exploration for large scale embedded systems. The implementation integrates a mathematical expression evaluator for the constraint model, and defines requirement sets using a database core. The derived constraints may be reviewed by the designer and adjusted manually if necessary.

During an exploration exercise the automatic constraint derivation is playing a very important role. By defining ten alternative architectures and five different requirement sets, for instance, we are able to evaluate up to fifty different scenarios with ease. These scenarios may include "what-if" questions, scalability tests for future extensions, flexibility and stress conditions etc. The power of the constraint derivation is such that if a requirement set for one of these conditions is defined, it can be reused to generate constraints for all the defined alternatives of the system.

As a driver application for our tool set we used the LOFAR radiotelescope, which is a good example of a large-scale distributed embedded system. The tool has been used to explore different realizations for the digital signal processing station of LOFAR[9].

The figure 8 depicts the constraint derivation and propagation for the beam-forming application of the DSP station. The application is specified as a Kahn process network[10], which is partitioned by the use of containers, shown in the first section of the picture. Initially the constraints were only derived on the components of the beam-forming cluster (shown in the second section), but after propagation they were propagated through the hierarchy all the way up to the containers.

During the exploration we did not encounter errors in the validation procedure - which is normal, if we are to consider that this is a real-life application. However, the error detection and adjustment were verified using small artificial examples during the development of the tool.

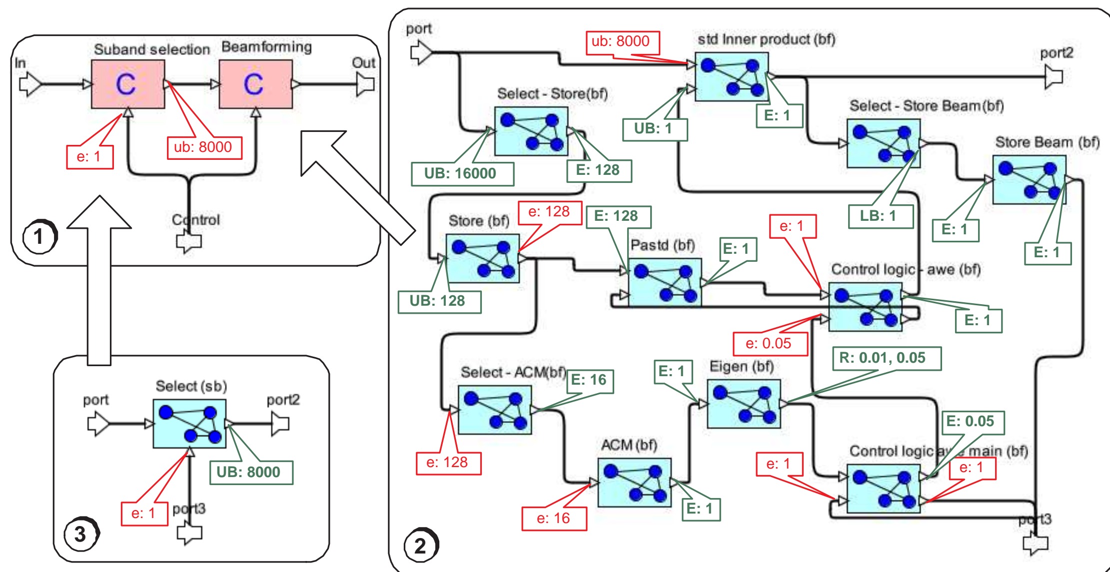

Fig. 8. A real-life example of the constraint derivation: the beam-forming subsystem of the LOFAR digital signal processing station. The application containers(1) partition the process network into a sub-band selection cluster(2) and a beam-forming cluster(3)

VII. Conclusions

The translation of requirements into constraints is a crucial step in the design of any system. When fixed requirements are given, the translation usually has to be performed only once before the optimization stage. However, when the requirements are constantly changing, a need for automating the translation process becomes apparent. This is especially true for large scale embedded systems, which undergo extensive exploration in the initial development stages.

In this paper we have presented structured methods of automatically deriving constraints from user requirements and propagating these constraints through a hierarchical system topology. The derived constraints are also checked for inconsistencies and adjusted where possible.

We have implemented these methods in a software platform called the *Massive Exploration Tool*. This tool has been successfully used during a real-life exploration exercise for the digital signal processing station of the LOFAR radiotelescope.

References

[1] L. Nicolae, S. Alliot, and E. Deprettere, “Component-level design space exploration for large scale embedded systems,” in *In Proc. Advanced School for Computing and Imaging (ASCI’03)*, pp. 68–73, June 2003.

[2] S. Alliot, *Architecture Exploration For Large Scale Array Signal Processing Systems*. PhD thesis, Leiden University, 2003.

[3] G. Taguchi, *System of Experimental Design*, vol. 1,2. New York: UNIPUB/Krass International Publications, 1987.

[4] F. Balarin, E. Sentovich, M. Chiodo, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli, *Hardware-Software Co-design of Embedded Systems - The POLIS Approach*. Kluwer Academic Publishers, 1997.

[5] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, “Metropolis: An integrated electronic system design environment,” in *IEEE Computer*, no. 1 in 36, pp. 45–52, 2003.

[6] S. Alliot, “A performance/cost estimation model for the large distributed array signal processing system and specification,” in *In Proc. Workshop on Synthesis, Architectures, Modeling, and Simulation (SAMOS03)*, pp. 154–160, July 2003.

[7] F. Balarin, J. Burch, L. Lavagno, R. Passerone, A. Sangiovanni-Vincentelli, and Y. Watanabe, “Constraints specification at higher levels of abstraction,” in *In Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT2001)*, 2001.

[8] S. Alliot, M. van Veelen, L. Nicolae, and A. Coolen, “The MASSIVE Exploration Tool.” University Booth Demonstration, Design Automation and Test in Europe (DATE’03), Mar. 2003.

[9] S. Alliot, “LOFAR station digital processing, architectural design description,” Tech. Rep. LOFAR-ASTRON-ADD-007, ASTRON, October 2002. www.lofar.org.

[10] G. Kahn, “The semantics of a simple language for parallel programming,” in *Proc. of the IFIP Congress 74*, North-Holland Publishing Co., 1974.